CS 159

C Programming Applications for Engineers

Spring 2026

Purdue University

STIPES PUBLISHING L.L.C.

CS 159

C Programming (Applications for Engineers)

Spring 2026

William Crum
Department of Computer Science
Purdue University

ISBN: 978-1-64617-784-4

Copyright © 2026
Stipes Publishing L.L.C.

Published by

204 West University Avenue
Champaign, Illinois 61820

CS 159: C Programming (Applications for Engineers) – Spring 2026
ISBN 978-1-64617-784-4

Copyright © 2026, Stipes Publishing L.L.C.
All rights reserved.

Permission in writing must be obtained from the publisher before any part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, or by any information storage or retrieval system.

This copyrighted text licensed to [redacted] on [redacted] via [redacted]

Table of Contents

Topic	Page
Syllabus	1
Course Programming and Documentation Standards	9
Lecture Notes	19
Chapter 1 – Introduction to Computers	21
Chapter 2 – Introduction to the C Language	31
Chapter 3 – Structure of a C Program	43
Chapter 4 – User/Programmer Defined Functions	53
Chapter 5 – Selection	72
Chapter 6 – Repetition	92
Chapter 8 – Arrays	116
Chapters 9 and 10 – Pointers and Pointer Applications	147
Chapter 11 – Strings: Character Arrays	155
Fall 2025 Exams and Solutions	165
Fall 2025 Midterm Exam #1	167
Fall 2025 Midterm Exam #2	179
Fall 2025 Final Exam	192
ASCII Table	209
Operator Precedence Table	210

This copyrighted text licensed to on via

This copyrighted text licensed to on via

CS 159 – C Programming (Applications for Engineers)
Spring 2026 Syllabus

Course Staff

Instructor:	**Instructor Office Location:** **Building:** **Room:**	**Instructor Office Hours:**

Please visit during instructor office hours for any administrative concerns regarding the course.

- If you have a concern then it is expected that you make the effort to visit office hours of your instructor or arrange another time to meet by discussing alternatives with them after lecture.

TA Office Hours

Course Motivation and Objectives

CS 159 introduces the tools of software development that have become essential for innovative and creative problem solving in science and engineering. Educators and employers agree that it is important for future technical professionals to be able to function as part of a technical team and develop the ability to effectively communicate complex concepts. CS 159 will require students to work in assigned teams for lab assignments. Educational research informs us that structured collaboration leads to increased learning gains for all students participating in an introductory programming course. **Collaboration is a requirement of the course.** You will be assigned to your teams by your instructor.

CS 159 explores the concepts of programming using a language and development environment that are new to most students. The primary objective of the course is to help students recognize how programming concepts are common across a variety of languages to which you are exposed in your undergraduate education and how those concepts can be used to solve a problem.

Course Prerequisites and Preparation

CS 159 is an introductory programming course without official prerequisite but does assume mathematical and physical science knowledge typical of a first-year engineering student.

The University expects students to invest 6-9 hours per week in preparation, in addition to time spent in class, for a three credit hour course. Previous students in CS 159 have reported that their keys to success included; attending lecture, active participation in lab, preparation and regular review of course materials, and planning to start and complete all assignments well in advance of the due date.

Course Required Materials

- Computer Science, A Structured Programming Approach Using C, Forouzan and Gilberg, THIRD EDITION, ISBN: 978-0-534-49132-1
  - An e-book alternative is available: https://www.cengage.com/shop/isbn/9781337907651
- Programming Applications for Engineers Course Packet (Spring 2026 edition)
  - Physical copy available only in West Lafayette bookstores. See Brightspace for a link to purchase an electronic version.
- Vocareum License, instructions will be provided on Brightspace to complete purchase.

Learning Resources

- All relevant class information, updates, and announcements will be available on the Brightspace site https://purdue.brightspace.com/.

This copyrighted text licensed to on via

Important Dates

Midterm Exam #1	Midterm Exam #2	Final Exam
Date: **Time:** **Location:**	**Date:** **Time:** **Location:**	**Date:** **Time:** **Location:**

- The final exam **may** be on Saturday May 9, 2026. Requests for alternative exam offerings will not be considered for reasons other than those outlined by the regulations of the university. **Please make your end of semester travel arrangements accordingly.**

Grading

Assignment	Points
Homework	35
Lab Tasks	60
Lab Quizzes	65
Midterm Exams	200
Final Exam	150
Lecture Quizzes	40
Total Possible:	550

Grades

Grade	Points Required
A	470
B	415
C	360
D	305

The instructor reserves the right to lower the minimum score required for each letter grade. If any change is made it will not be announced until **after the final exam**.

- At no time during the semester will it be speculated if this will be done or how much any given cutoff will be lowered. You should have no expectation that all cutoffs if moved will be moved by an equal amount.
- The use of plus (+) and minus (−) additions to a grade will be considered for those few individuals who are very close to, but fall short of, one of the posted minimum scores.

Attendance

You are expected to attend every lab this semester. To be eligible to retain points earned on a lab programming assignment and related lab quiz you **must be in your assigned lab within 5 minutes of the start of your lab session**. Your lab instructor will provide the instructions necessary to register your attendance. Labs include collaborative efforts and it is an unfair expectation that your lab partners or teaching assistant incorporate a student arriving late, or who is absent, into the team effort. **Students late or absent to more than two lab sessions will fail the course.**

You are expected to attend every lecture this semester. Seats will be assigned in the lecture hall and attendance taken to ensure that only those students officially enrolled and physically present in each section are eligible to earn points for lecture quiz participation.

Emergency Absences and Make-Up Work

University policies on absence and absence reporting are outlined by the Office of the Dean of Students: https://www.purdue.edu/advocacy/students/absences.html

The instructional staff of CS 159 will require students to submit relevant forms through the Office of the Dean of Students for absences outlined by the University Senate including grief absences, jury duty absences, emergent medical absences, and military training absences.

Remaining **documented and serious hardships** may be considered for missed work. When you have **documentation** of what you consider to be such a hardship then you must contact the instructor in a timely manner **during office hours when you are able to resume participating in class**. Any student who knows in advance of an absence must make a request for consideration one week prior to the absence. Make-up requests for **medical reasons that do not fall under the policies linked above MUST be accompanied by documentation from a medical professional** stating the dates you were under their care and the date you were cleared to return to school/work.

This copyrighted text licensed to on via

Assignments

Exams (two evening midterms 100 points each, one final exam worth 150 points)

Exams will be **individual** assessments of your knowledge. Exams will consist of multiple-choice problems covering programming concepts, best programming practices, previous programming assignments, and the interpretation of code.

Old exams are provided in the course notes packet for your preparation. Please read the cover page of the previous exam section (page 165) for more information on how old exams can be a useful part of your comprehensive preparation process.

Homework Assignments (7 total, 5 points each)

The homework assignments are **individual** efforts designed to give you the opportunity to solve programming problems on your own **without the assistance of other students**.

Most assignments will be posted on Brightspace the day following the due date of the previous assignment.

Please review the course policies as they relate to **academic integrity and the use of artificial intelligence** found later in this document.

Homework Assignment	Due at 11pm on
1	January 26
2	February 9
3	February 23
4	March 9
5	March 30
6	April 13
7	April 26

Lab Tasks (12 total, 5 points each):

Lab assignments are to be completed **collaboratively** in your assigned lab groups and each of these lab programming assignments will be due 30 minutes prior to the next time your lab section meets (see schedules later in this document).

- Collaborative groups are expected to communicate who will submit the assignment, when the assignment will be submitted, and how progress will be confirmed with all participating group members.
- Setting expectations for every member of the group will improve the likelihood that a complete assignment will be submitted.
- Lab partners failing to participate and contribute to the satisfaction of all group members will not receive credit for the lab assignment.
- Lab teams will be reassigned after lab #4 and lab #8 during the semester. Enrollment and course participation may warrant additional changes.

Weekly Lab Schedule

Week of	Lab Assignment	Week of	Lab Assignment
January 12	Lab + Quiz #0	**March 9**	Lab + Quiz #8
January 19	Lab + Quiz #1	**March 16**	**Spring Break**
January 26	Lab + Quiz #2	**March 23**	Lab + Quiz #9
February 2	Lab + Quiz #3	**March 30**	Lab + Quiz #10
February 9	Lab + Quiz #4	**April 6**	Lab + Quiz #11
February 16	Lab + Quiz #5	**April 13**	Lab + Quiz #12
February 23	Lab + Quiz #6	**April 20**	Lab Quiz #13
March 2	Lab + Quiz #7	**April 27**	NO **LAB**

This copyrighted text licensed to on via

Lab Quizzes (13 total, 5 points each)

At the end of each lab meeting there will be an **individual** assessment of your knowledge related to the topics introduced in lecture and implemented in the most recent lectures and assignments. Knowledge of course standards and good programming practices will be evaluated throughout the semester.

- The best way to prepare for quizzes is to be present in lecture and actively participate with your team during the process of solving the lab, including contributing to both the written problems and the development of the programming problem solution.
- Lab quizzes can only be completed from your official lab location while being proctored by your lab instructor. **No resources are permitted for use during the quiz.**

Lecture Quizzes (40 points possible, lowest two scores dropped)

At every meeting of lecture, starting with the second week of the term, a lecture quiz will take place. Quizzes may occur at the start, in the middle, and/or at the end of lecture. **It is a requirement to be in your assigned seat to retain any credit earned on a lecture quiz.**

The ability to access iClicker with your Purdue credentials is **required** to participate in lecture quizzes. You should bring your selected device to every lecture. Please review the **academic integrity policies** regarding the **misrepresentation of identity or location** as it relates to participating in a lecture quiz for another student.

The lowest two lecture quiz scores of the semester will be dropped and only absences for reasons outlined by the University Senate (see page 2) can be considered when requesting consideration for missed lecture quizzes beyond the two which are dropped.

Lecture Schedule

Week of	Tuesday	Thursday	Week of	Tuesday	Thursday
January 12	Introduction	Chapter 2	**March 9**	Chapter 6	
January 19	Chapter 2		**March 16**	**Spring Break**	
January 26	Chapter 3		**March 23**	Chapter 8	
February 2	Chapter 4		**March 30**	Chapter 8	
February 9	Chapter 4		**April 6**	Chapter 9/10	
February 16	Chapter 5		**April 13**	Searching and Sorting Arrays	
February 23	Chapter 5		**April 20**	Multidimensional Arrays and Recursion	
March 2	Chapter 6		**April 27**	Midterm Compensation – **No Lectures**	

Grading Expectations and Policies

Our expectation of your lab instructor is that they grade your assignment in a **timely manner and provides you with adequate feedback** for improvement. If you feel this is not the case please address your concern to your instructor. Typically, your lab instructor should complete their final evaluation of programming assignments 4-6 days after each is due.

To request a re-grade on any assignment you must deliver your **request in writing to the office hours of your instructor**. You have five days to appeal any grade from the day the score of the assignment is posted to Brightspace. After that period the grades are frozen and no further appeal will be considered. A re-grade is not a second chance to complete an assignment, neither is it a means to challenge assignment requirements, course policies, or programming and documentation standards.

A re-grade request must include the following:

1. A physical copy of the original graded assignment (if applicable).
2. An attached description of why you believe reevaluation of your effort is warranted.

This copyrighted text licensed to on via

Important Assignment Guidelines

All assignments must comply with **the programming and documentation standards of the course.** Programs that compile **and** meet minimum assignment requirements but are not logically correct or complete may be considered for partial credit. **To receive full credit, your program must (1) produce correct results, (2) be well-designed, (3) make efficient use of the limited resources of the computer, (4) follow assignment requirements, and (5) adhere to course programming and documentation standards.**

An assignment that is not submitted as expected cannot be considered for a grade. Only work submitted correctly prior to the assignment deadline can be considered for grading. **Late work is not accepted.**

Demand for resources and course staff will increase as an assignment deadline nears. Waiting until the last minute to work on your programming assignments is discouraged! Course policy is **NOT to extend** deadlines unless official course resources are unavailable for an extended period near the deadline of an assignment. You are responsible for understanding how to use the technology and tools utilized within the course.

Establish a habit of submitting every programming assignment early, this practice will allow sufficient time to seek assistance should you experience any difficulties solving or submitting an assignment.

Academic Integrity

The issue of academic integrity is taken seriously within CS 159. The consequences for violating course policies are significant both within the course and by the disciplinary arm of the Office of the Dean of Students.

You are encouraged to discuss any CS 159 topic including high-level ideas about how to approach a programming assignment. However, under **no circumstances will exchange of, or shared access to, code via written or electronic means be permitted** between teams for collaborative assignments or individuals for individual assignments.

It is considered dishonest either to read another solution or to provide anyone with access to your work (or that of another student). Be mindful when working through code with others on individual assignments as this is discouraged. The work you submit must be your own original effort and not the result of unacceptable, even if unintentional, collaboration.

Why enforce academic integrity? Academic integrity violations equate to theft. Theft of the work and time from the individual who developed the solution and theft of the time of University staff to conduct an inquiry into such matters. Ultimately, it amounts to **theft from every student** who has come to Purdue University, made a significant financial investment in their education, and has the expectation that their degree will be valued by employers and other academic institutions upon completion.

When is it no longer acceptable to discuss an assignment with another student or someone not from my group?

- Discussions with peers are most appropriate during the early phases of solution development. Once you begin to design and implement (write code or develop specific logic) your solution you should be referencing course staff members **exclusively** for assistance.
- Working closely with another student on a homework assignment may result in highly similar work due to collaboration. Collaboration may not have been the intended approach to solving the problem but the end result of working closely with others for extended periods of time.

Every student is responsible for protecting their own work. Do not make the assumption that roommates, neighbors, significant others, or other **"trusted"** individuals would not take advantage of knowing your password, having access to your computer, or taking a picture of your work when left on an unattended machine. **You are responsible for such events that leave your work unprotected.**

Many free on-line compilers, collaborative editors, and file depositories will store your work and may leave it publicly visible unless you pay for upgraded services. Please understand the policies of any such service before you make use of them. **See the note above regarding your responsibility to protect your work.**

This copyrighted text licensed to on via

Academic Integrity (continued)

Do not make the mistake of thinking that **superficial changes in a program** (such as altering comments, changing variable names, interchanging statements, or additional white spaces) will avoid detection. If you are unable to complete the work yourself, it is unlikely that you will succeed in disguising the work of another as your own. We are adamant that **violations in any form will not be tolerated**. Even the most trivial assignment is better left without submission than if you violate course integrity policies to complete it.

As easy as it is to share an electronic copy of a file, to gain access to a file through account sharing, sharing a hard copy of your work, or seeking assistance from strangers on the Internet, it is as just as easy to analyze and detect such sharing as it results in similar efforts being submitted.

Internet forums, including premium services that advertise assistance in college courses, are monitored by course staff members. Any relevant code found is included with student submissions to test for similarity. Searching for and using solutions to assignments, requesting assistance on assignments, and the posting of assignments in a venue such as these is a **violation of course academic integrity policies and potentially violates University copyright**.

Assume that every final submission you make during the semester will be analyzed by at least one software similarity service. Similarity services will measure the percentage of similarity between your solution and those submitted by others in the course. Additionally, these services indicate the percentage and number of lines matched among submissions. **You will be solving problems this semester that have no single solution and your solution is expected to be uniquely yours.** Concerns regarding any of our policies should be addressed during office hours prior to the deadline of an assignment.

Usage of Generative Artificial Intelligence/Large Language Models. Tools such as these are intriguing but are a poor fit within a course attempting to teach novice programmers the fundamental topics and problem-solving techniques of computer programming. **The use of AI-generated logic in any portion of an assignment will be considered to be in violation of academic integrity expectations of the course.**

Minimum consequences for violating course policies will include:

- First offense, a **zero for the assignment**, a **reduction of one letter grade at the end of the semester**, AND a **referral to the Office of the Dean of Students** for disciplinary action.
- Second offense, a zero for the assignment, a **failing grade for the course**, AND a referral to the Office of the Dean of Students for disciplinary action.

Notable exceptions to the first offense minimum consequences:

- Any violation on an exam will result in a failing grade for the course and a Dean of Students referral.
- Acts such as a misrepresentation of identity or location will result in a failing grade for the course and a Dean of Students referral.
- Posting to, requesting from, or accessing solutions found on unapproved sources, particularly those found on-line, will result in a failing grade for the course and a Dean of Students referral.

Best Practices	**How to respond to a request from a peer for assistance on a programming assignment?**
• Develop all of your code on Vocareum and make numerous submission as you progress. This will provide a detailed history of your efforts. • Utilize software connected to your career account when developing tools used to plan your solution to a programming problem.	• Refer the student to official sources of help. • Encourage the student to begin assignments earlier and to reflect on why their efforts in the course are not resulting in the success they hope to obtain. • Remind the student of academic integrity concerns.

This copyrighted text licensed to on via

Collaborative Learning/Teaming/Participating as a Member of a Technical Team

Here are our expectations of you and your group:

1. **Make time to meet with your group regularly.** There are 168 hours in a week, finding some common time for a group to meet should not be difficult. It is acceptable for part of the group to meet some of the time if everyone cannot participate in every meeting. Ultimately, each individual is responsible for planning and documenting their contribution to the group effort accordingly.
    - Reflect on the roles and responsibilities you had on each lab programming assignment. Recognize when you are not being utilized in a way that contributes to a better understanding of course content.
2. **Allow everyone an opportunity to express their ideas** on how to approach an assignment. One benefit of collaborative teaming is that everyone brings a different set of skills to the group and the resulting effort often is a stronger one than if it was completed individually.
    - When a group member becomes unresponsive to requests to meet or fails to update the other members of the group then those contributing members must prepare to continue without the unresponsive member.
3. **All group members must be satisfied with the final submission.** It is not acceptable for a group to submit an assignment that is not approved by all group members. "It is good enough" may be true for you but it is unfair for the others in the group who aspire for the highest grade possible.
    - Likewise, each group member must be satisfied with your contribution to the group effort.
4. **Each group member must fully understand the entire assignment submitted.** Do not start your group meetings by trying to delegate the tasks to the different group members. Everyone must understand and contribute to every aspect of the assignment and its development.
    - Assignments are an opportunity for you to demonstrate your knowledge of the programming concepts being utilized. Use assignments to serve as a measure of what you do and do not know and how to direct your future efforts in the course.
5. **Designate who will turn in the assignment, when it will be turned in, and how successful submission will be communicated with the rest of the group.** Set a goal to submit the assignment well in advance of the due date to avoid any last minute problems.
    - Have a plan, including a meeting just prior to the assignment deadline, to deal with unexpected challenges completing your solution. It is easy to cancel this meeting when it is not needed.
6. **You will work with the group assigned.** Please contact us with concerns you may have with your group. Take a professional approach with your group experience as similar to what you may experience at an internship or co-op experience.
    - Future group assignments will take into consideration lecture attendance and assignment completion as an indicator of your interest in participating in the course. Active students in the course should not be burdened with partners who are not willing to stay engaged with course content.

Nondiscrimination Statement

Purdue University is committed to maintaining a community which recognizes and values the inherent worth and dignity of every person; fosters tolerance, sensitivity, understanding, and mutual respect among its members; and encourages each individual to strive to reach his or her own potential. In pursuit of its goal of academic excellence, the University seeks to develop and nurture diversity. The University believes that diversity among its many members strengthens the institution, stimulates creativity, promotes the exchange of ideas, and enriches campus life.

Students with Disabilities

Purdue University strives to make learning experiences as accessible as possible. If you anticipate or experience physical or academic barriers based on disability, you are welcome to inform your instructor to discuss options. You are also encouraged to contact the Disability Resource Center at: drc@purdue.edu or by phone: 765-494-1247.

Students with an existing letter of accommodation are to refer that letter to their instructor and follow up with an e-mail to initiate a conversation regarding how your accommodations may be implemented in the course.

This copyrighted text licensed to on via

Emergency Preparation

In the event of a major campus emergency, course requirements, deadlines, and grading percentages are subject to changes that may be necessitated by a revised semester calendar or other circumstances.

- In case of a campus emergency, check Brightspace for announcements. Follow all official university issued instructions. See https://www.purdue.edu/ehps/emergency_preparedness/ for more information.

Mental Health/Wellness/Basic Needs Statement

Purdue University is committed to advancing the mental health and well-being of its students. If you or someone you know is feeling overwhelmed, depressed, and/or in need of mental health support, services are available. For help, such individuals should contact Counseling and Psychological Services (CAPS) at 765-494-6995 during and after hours.

If you need support and information about options and resources, please contact the Office of the Dean of Students. Call 765-494-1747.

Disclaimer

- **The instructors reserve the right to revise the syllabus and will provide notice through a variety of sources such as lecture recordings, Brightspace announcements, and e-mail.**

This copyrighted text licensed to on via

COURSE PROGRAMMING AND DOCUMENTATION STANDARDS

Programming Standards – Includes definitions of good programming practices.

Documentation Standards – Formatting and documentation expectations.

Please read and revisit the section carefully throughout the semester.

This copyrighted text licensed to on via

This copyrighted text licensed to on via

Course Documentation and Programming Standards

Rationale

Programming is just one skill that is part of the larger discipline of software engineering. Like other engineering disciplines, decades of history have been distilled into a set of best practices that become an expectation for new practitioners. At this time there is no legal basis for licensure in software engineering, but consider licensure requirements in other engineering fields: agricultural and biological, chemical, civil, computer, electrical, industrial, mechanical, and nuclear engineering all have professional engineering standards that aim to protect public safety.

C is an inherently unsafe language. Much contemporary research has been devoted towards mitigating the more dangerous aspects of the language, either by enforcing standards, creating safe subsets of the C language, or developing completely new programming languages. Because this course is titled *C Programming*, standards will be emphasized over other considerations.

Most companies have extensive documentation requirements in order to facilitate the transfer of previously developed code from one employee to another without the need to invest a great deal of time in the effort necessary to comprehend the work of the original developer.

This is the authoritative document for standards in this course. You may consider it to be a subset of common programming standards that serve a safety purpose, have instructional value, or promote consistency. The course notes and examples in lecture will attempt to abide by these standards as closely as possible to serve as an example. However, to save space in the notes and time during lecture, some standards may be excluded; this is not an option for your assignments. Additional resources approved by the course, including the textbook, do not abide by all of our standards. Moreover, references external to the course will often vary wildly from the anticipated and required practices of the course.

Grading Expectations

All of the expectations and standards in this document must be followed to earn all points possible on programming assignments. Points on an assignment will be deducted if your code and/or logic is difficult for your lab instructor to evaluate because it fails to meet the expectations outlined in this document. We reserve the right to assign a score of zero for the infraction of any of these expectations.

For some students this will require breaking previously developed programming habits. What makes a habit or programming practice less desirable? Anything that makes programming harder for beginners, increases the vulnerability of software to hacking or crashing, unnecessarily creates complex code, or represents obsolete features of the language. The first point is particularly important: you will be working in teams with students who have never programmed before. All students on a team must understand the work that is being submitted. Scores may be adjusted if any discrepancies are discovered.

Every assignment will attempt to provide sufficient detail on all requirements. However, because the problems solved in this course have no unique solution, you will always be evaluated according to the following general criteria:

1. Conform to all assignment-specific requirements provided in the instructions.
2. Abide by all course programming and documentation standards.
3. Implement the appropriate logic to solve the problem, including accepting input in the order expected and producing the expected output.
4. Follow good programming practices.
5. Make efficient use of the limited resources of the computer.

It is your responsibility to ensure compliance with these general expectations as well as all specific standards in this document. Some automated feedback will be provided each time you submit a programming assignment, but by no means is this an exhaustive review of your work. Your lab instructor and your lecturer (in accordance with this document) have the final say on all aspects of your programming assignment grades.

You may ask course staff members for specific advice in making your work compliant with these standards, but “pre-grading” is not an option due to the number of students enrolled.

This copyrighted text licensed to on via

Documentation and Style

The requirement of good documentation practices is universal among technical professionals; the primary motivation of having documentation standards is to make your code, which ultimately represents your logic, as easy to understand and easy for your grader to evaluate as possible. The solutions you are in the process of developing must be quickly interpreted when seeking assistance both in this class and from your colleagues, supervisors, and clients in the future. Finally, documentation is an opportunity for you to enhance your understanding of code you or your team members have written.

Style, while less important than documentation, nonetheless enforces consistency and therefore aids readability.

1. Assignment Headers

- Assignment headers are used in the course to provide a detailed description of the logic being implemented in a program or user-defined function.
- There are three total headers we will use this semester. One for labs, one for homework, and another for user-defined functions (in both labs and homework).
- The academic integrity and artificial intelligence (AI) statements are provided for you and MUST remain in the header. Do not make any changes to either statement.
- The alternating characters at the beginning and end of each assignment header are part of a security and identification feature and must not be modified.
- A verbose program description is necessary. If your lab instructor is unable to follow your logic after a quick glance at your code then they should be able to determine from your description the algorithm you were attempting to implement. A verbose program description will mean fewer necessary comments within the code of your program.
- With functions, it is important to comment your parameters (those values received by a function) in the header! There is insufficient space to comment parameters in the first line of the function definition and it must be included here within the header. Add or remove lines of parameters as needed.

Lab Assignment Header: Inserted using `vi`'s `:hlb` command inside Vocareum.

```
/*****+--*--***-*-----*-****-*--*--*-***-*---*-*************************
*
*  Lab #:
*
*  Academic Integrity Statement:
*
*  We have not used source code obtained from any other unauthorized source,
*  either modified or unmodified. Neither have we provided access to our code
*  to another. The effort we are submitting is our own original work. We have
*  not made use of any AI generated code in this solution.
*
*  Program Description:
*
******+--*--***-*-----*-****-*--*--*-***-*---*-*************************/
```

Homework Assignment Header: Inserted using `vi`'s `:hhw` command inside Vocareum.

```
/*****+-**--***-*-----*-****-*--*--*-***-*---*-*************************
*
*  Homework #:
*
*  Academic Integrity Statement:
*
*  I have not used source code obtained from any other unauthorized source,
*  either modified or unmodified. Neither have I provided access to my code
*  to another. The effort I am submitting is my own original work. I have not
*  made use of any AI generated code in this solution.
*
*  Program Description:
*
******+-**--***-*-----*-****-*--*--*-***-*---*-*************************/
```


This copyrighted text licensed to on via

Function Header: Inserted using `vi`'s `:hfx` command inside Vocareum.

```
/*****+-**--***-*---*-*---------**------**********************************
*
*  Function Information
*
*  Name of Function:
*
*  Function Return Type:
*
*  Parameters (list data type, name, and comment one per line):
*    1.
*    2.
*    3.
*
*  Function Description:
*
******+-**--***-*---*-*---------**------**********************************/
```

2. Identifiers

- We will consider the following to be objects in this course: variables, functions, and symbolic/defined constants. You must give your objects identifiers that reflect their purpose in the program.
- Rarely is a single character identifier for an object meaningful. A reasonable exception would be for a loop control variable used for simple counting or to access the elements of the array.

`int x;`
`int y;`
`int z;`
	Can you determine what the variables `x`, `y`, and `z` accomplish for the program based only on the name of the variable? Perhaps they hold information on a 3D point? If the program contains no 3D coordinate data, then the use of `x`, `y`, and `z` as variable names is unclear.

`int numStudents;`
`float objectMass;`
`char courseGrade;`
	Relevant variable, constant, and function names assist in the documentation and readability of the logic you are attempting to implement.

- Identifiers for variables and functions must consist of a mix of lower- and uppercase letters, digits, and underscores. The `snake_case` and `camelCase` conventions are equally acceptable so long as the choice is consistent in your work.
- Identifiers for symbolic/defined constants should be `ALL_UPPERCASE`.
- Identifiers beginning with an underscore character are often reserved by the language and should be avoided.

3. Commenting

- EVERY variable that you declare must have a brief comment to describe its purpose in the program!
  - Declare only a single variable per line. This leaves room for you to include a comment to the right of each declaration.
- Within a given function it is expected that the occasional complex segment of code be documented in order to inform others regarding that logic you are attempting to implement.
- Comments should not reiterate what is already obvious from the code. A good rule of thumb is comments should describe the WHY, not the HOW.
- The auto-grader will look for a particular ratio of comments per lines of code. You are expected to satisfy this requirement. A deduction for an "insufficient number of comments" may indicate that your solution is overly complex or contains duplicate code that can be simplified. If not, add documentation stating otherwise.

4. Source File Organization

- All code must be written in the following order: 1. assignment header, 2. preprocessor directives, 3. function declarations (which do not need to be commented), 4. `main` function (which does not need a declaration or header), 5. function definitions (each with their own function header).

This copyrighted text licensed to on via

5. Constants

- Named constants are expected to be implemented using the `#define` preprocessor directive (i.e. a symbolic/defined constant). Variables declared with the `const` type qualifier are not permitted.
- Define all symbolic/defined constants at the top of your program immediately following the assignment header.
- Symbolic/defined constants need not be commented.
- Maximize your use of symbolic/defined constants and minimize your use of literal constants.

6. Indentation and Brace Positioning

You must indent all code inside the body of a structure two (additional) spaces. Rarely should you ever begin a line of code in the first column of your editor (except for preprocessor directives, headers, and function declarations and definitions).

Correct

```
for(x = 1; x <= MAX; x++)
{
  printf("%d", x);
}
```

Correct

```
if(y == 3)
{
  for(g = 0; g > MIN; g--)
  {
    y = y - g;
  }
}
```

Correct

```
do
{
  e = m * c * c;
  avg = sum/number;
}while(r == 0);

// Note: while on same
// line as closing }
```

Correct

```
if(x == 3 || y == 2)
{
  z++;
  a = a - 1;
}
```

INCORRECT

```
if(x == 3 || y == 2)
  {
    z++;
    a = a - 1;
  }
```

INCORRECT

```
if(x == 3 || y == 2){
  z++;
  a = a - 1;
}
```

Switch constructs will be indented as seen in the textbook; see page 258.

7. Spacing and Blank Lines

- Place an additional line between sections to help group the code by logic related to the same task.
- Balance the use of additional lines and failing to place any throughout your code.
  - There is no need to double space all lines of code or to include multiple consecutive blank lines.
  - Like comments, the auto-grader will look for a particular ratio of blank lines per lines of code. You are expected to satisfy this requirement. A deduction for an "insufficient number of blank lines" may indicate that your solution is overly complex or contains duplicate code that can be simplified.
- It is a requirement to place a space between binary operators and their operands, such as: `y = y - g;`

8. Keeping Local Declarations Separate from Executable Statements

- The compiler in this course has been configured to require that all variables in a function are declared prior to the first executable statement. See Figure 2-2 in Chapter 2 of C programming text for more information.
- Originally a technical limitation of the language, this separation continues to be enforced in the course as a matter of style that serves two purposes: 1. to aid the developer and the reviewer in understanding the amount and kinds of variables in use within a given function by collecting them in a common location, and 2. to centralize the documentation of all such variables in that same location.
- The same reasoning applies to the declaration of loop control variables within the first expression of a `for` loop. The compiler in this course has been configured to prohibit it so that you are required to declare these variables in the same location as all other local variables.

9. Multiple Assignments/Statements

- Do not put multiple assignment expressions (`x = y = z`) or statements (`a; b; c;`) on a single line of code.

This copyrighted text licensed to on via

Programming Standards

1. Variable Initialization

```
int numStudents = 0;                                    //OKAY TO KEEP HERE
float objectMass = calcMass(acceleration, velocity);    //PLACE AMONG EXECUTABLES
```

- It is acceptable to initialize a variable declared in the local declaration section of a user-defined function to a constant value. More complex initializations are expected to be completed within the executable statements section of the function.

2. Scope

- All variable declarations must appear in the local declaration section of a function. Variables are **never** permitted to have global scope. This requires good function design as well as reinforces the techniques of parameter passing and return values.
- All function declarations will be global, to be placed between the preprocessor directives and the `main` function.
- Do not reuse an identifier between two objects that have overlapping scope (MISRA C: 2023, Rule 5.3).
- All variables, parameters, and function return values must have an explicit data type (MISRA C: 2023, Rule 8.1).

3. Bodies and Braces

- It is possible in certain situations to omit the braces (`{` and `}`) around the body of structures such as `if`, `for`, and `while`. However, the course will require these braces to be used even when unnecessary. Beginners have a difficult enough time pairing up their curly braces and a misplaced brace can be one of the more challenging errors to resolve.
- We will discuss the **dangling** `else` problem in chapter 5 (page 245 of the C programming text, figure 5-14) as an example of one difficult logical error to debug when you omit the braces we have required.

4. Standard Libraries

- You are permitted to use many of the functions in the standard C libraries introduced in class. It is your responsibility to ask a course staff member about a function or library we have not discussed in class that you would like to use.
- Often, the additional requirements or problem description of an assignment will indicate when a particular library is permitted or prohibited for a given programming problem.

5. Repetition

- Do not compensate with redundant code or meaningless variable initialization when selecting a pretest loop when a post-test loop was the proper choice for a repetitive process.
- Use `for` loops only with counter-controlled processes (MISRA C: 2023, Rule 14.2).
- Make use of all three expressions with every `for` loop. Do not leave expressions empty or place meaningless statements in any of the expressions such as `x = x` (MISRA C: 2023, Rule 14.2).
- The comma operator must not be used (MISRA C: 2023, Rules 12.3 and 14.2).
- The use of infinite or perpetual loops (MISRA C: 2023, Rule 14.3) would violate the expectation that control-forcing statements are prohibited; refer to the next section.
- Recursion must only be used with counter-controlled processes. Without knowing the total number of repetitive function calls, it is possible for the program to crash in an irrecoverable way.

6. Control-Forcing Statements

- You will observe control forcing statements being used in the C programming text (and other external sources) but their use is prohibited within the course.
  - `exit`, `goto`, `break` (permitted only with `switch` structures), and `continue` are prohibited. These statements complicate the understanding of a piece of code for a new programmer and can be avoided through restructuring.
  - Multiple `return` statements in the same function are prohibited (MISRA C: 2023, Rule 15.5).

This copyrighted text licensed to on via

7. User-Defined Functions

Upon introduction of user-defined functions (chapter 4) only the following will be permitted in the `main` function:

1. Declaration of those variables that need to be passed between functions.
2. Calls to user-defined functions.
3. A minimum amount of control structures (loops, selection) to maintain `main` as the control center of the program.

The `main` function is intended to be *the main function* of the program and a majority of the function calls should originate in `main`. Most of the data defined in the program should be declared in `main`. Anyone who reads only the `main` function of your program should be able to quickly understand the overall algorithm of your program.

Each function must have a single specific task that can be tested separately from the rest of your program. This is a subjective measure, and you are encouraged to seek advice from the course staff when designing your functions. A common rule of thumb in the industry is that each function definition should be not be longer than a single screen of text.

Passing individual values by address must meet the following criteria:

1. More than one value needs to be revised and that revision be retained in the calling function.
2. The function completes only a single fundamental task within the larger problem being solved.

The goal of these guidelines is to promote good function design and to reinforce the necessary concepts.

8. Arrays

- All arrays will be static (of fixed-length) until we introduce the topic of dynamic memory allocation in chapters 9 and 10 of the textbook.
- A symbolic/defined constant must be used when declaring the size of an array.
- Variable-length arrays are prohibited (MISRA C: 2023, Rule 18.8). Compilers often generate poor performance for these constructs, and they suffer from the same irrecoverable errors as that of unexpectedly deep recursion.
- Only a small aspect of dynamic memory allocation is introduced in CS 159 and the examples provided in notes packet represent the expectations and set the limits of what should be used in assignments. There is little time in the course to approach this topic in sufficient detail for new programmers, so this is left for subsequent courses.

9. Advanced Topics

- In all programming assignments we will specify from which chapters you may reference when developing your solution. The use of advanced topics (material not yet or never covered in the course) is expressly prohibited on all assignments.
- You are expected to demonstrate your mastery of the content covered in the course, not any content that has been learned prior to or outside of the course.
- We reserve the right to deduct all points and initiate an academic integrity investigation when advanced topics are implemented. These implementation details are more often an indication that the solution was developed by generative artificial intelligence or someone who is not enrolled in the course, both of which are prohibited by the syllabus.
- Common implementations of advanced topics include, but are not limited to:
  - Variable length arrays as discussed previously.
  - String variables or functions. Unless specified otherwise, the only permissible use of strings is as the first argument (format string) for calls to `printf` and `scanf`. This expressly prohibits the use of the `%s` placeholder.
  - The `const` type qualifier. As discussed previously, symbolic/defined constants should be used instead.
  - `void *` pointers or multiple levels of indirection. There is no valid reason to need these in the assignments.
  - The `qsort` standard library function. One of the goals of the course is to understand and implement simple sorting algorithms; using a standard library function defeats that purpose.
  - Standard library functions that consume input or produce output other than `scanf` and `printf`. All input and output requirements in this course can be implemented with just these two functions.
  - Standard library functions for dynamic memory allocation other than `malloc` and `free` as discussed previously.

This copyrighted text licensed to on via

10. Efficient Design

- One measure regarding the quality of ALL assignments is the implementation of efficient design. Students of innovative work in science and engineering should anticipate and appreciate that design is critical to the evaluation of any product including software.
- We require that you implement an efficient and logically correct algorithm as part of your assignment. In some cases this may make the distinction between letter grades on the assignment. To receive full credit on an assignment we expect correct logic AND good design.

Some guidelines related to the efficient use of the limited resources of the computer:

- Conserve memory.
  - Do not use multiple arrays (or dimensions) when one will do the job.
  - Do not declare arrays to be large beyond your data needs (specified by the assignment).
  - Do not use arrays when they are unnecessary.
- Conserve the effort of the processor.
  - Do not write nested loops for tasks that can be done in one (or none).
- Shorter code is usually better. Recognize patterns in your code. Avoid duplication of similar logic within a program.
  - If you find yourself repeating code in several places, create a function such that the code only appears once in your program and is called when needed.
  - Another common scenario involves several functions have been written that do essentially the same thing, or contain a considerable amount of similar code. In such cases you should attempt to consolidate as much as possible.
- Follow top-down design. Each function should have a well-defined task and not a series of tasks.
- Topics of efficient design will be commented on, where appropriate, in lecture.

11. Syntax Errors and Warnings

- Any submission with a syntax error in the source code will not be given consideration for partial-credit. You are responsible for testing your work thoroughly prior to submission.
- Compiler warnings will result in a loss of points and should be remedied prior to submission of your final effort. The compiler is a tool to assist in writing safe, efficient, and readable code: do not ignore any messages it produces.

12. Course Tools and Technologies for Developing Code

1. We DO NOT encourage that you develop, compile, or to test your code in ANY editor or compiler other than those found within Vocareum. Not all C programming language compilers are fully compatible. You should want to test your work under the same conditions that it will be processed and tested by the course.
2. It is your responsibility to thoroughly test your code within Vocareum as demonstrated during the first week of the semester prior to making the final submission of your work.
3. We will not support compilers, editors, and programming development environments other than those found on Vocareum.
4. You are advised to leave sufficient time prior to the assignment deadline to seek assistance should you encounter a difficulty with assignment submission. No late work will be accepted.

This copyrighted text licensed to on via

This copyrighted text licensed to on via

LECTURE NOTES

This copyrighted text licensed to on via

This copyrighted text licensed to on via

Hardware Basics

The four primary components of a computer:

1. A Central Processing Unit
2. Main Memory
3. Input/Output Hardware
4. Systems Interconnection - connecting all of the above

This copyrighted text licensed to on via

The following two hardware components of a computer are important to us as programmers:

1. The central processing unit or processor.
2. The main memory storage.

- **Why these two particular components?** An understanding of how the processor operates and what role the main memory plays for the programs we will write this semester will allow us to better utilize these **limited resources** of the computer.

The **processor (CPU)** is of interest as it is where all of the processing of **instructions and associated data** will take place. The CPU can be described as having four fundamental components:

1. Registers - A small amount of memory local to the processor used for temporary storage related to the current instruction being executed.
2. Arithmetic and Logic Unit - Hardware responsible for the calculations required of an instruction.
3. Control Unit - Coordinates the operations of the processor.
4. Interconnection - connecting all of the above.

This copyrighted text licensed to on via

Main Memory

- The memory of a computer will store the **data and instructions** of the programs that we will write this semester.
- You may have made the observation that the capacity of primary memory (RAM, 8-16 GB) on your personal machine is quite small in comparison to the size of your secondary memory (hard drive, 1 or more TB).
- Therefore, not every instruction or data associated with the currently executing program can be stored in primary memory, but, when instructions or data are needed, they must be **fetched, or retrieved,** from the hard drive and brought in to primary memory for use.
- Similar to addresses of homes on a street, the memory of the computer has addresses at which instructions and data can be located.
- In our programs we will select names that are meaningful to us as programmers rather than referencing a specific address in memory to represent and locate data.

Processing Instructions

- The CPU executes instructions using a process known as the **instruction cycle**.

Instruction Cycle Diagram:	
Why is secondary memory slower than primary memory? • Many secondary devices still have moving or mechanical parts. • Further physical distance from the CPU (or machine). • Increased time required to locate specific content in a high capacity device.	**If primary memory is faster then why not grow the capacity of this level and eliminate secondary memory?** • The current model is an economic one for consumers.

This copyrighted text licensed to on via

Memory Hierarchy:

This copyrighted text licensed to on via

Natural Languages versus Machine Languages

- When we describe problems that we wish to solve we do so in a **natural language**.
- **Natural languages** are spoken and have evolved over centuries in accordance with their usage.
- **Natural languages** are full of attributes that are unfit for use by a computer.

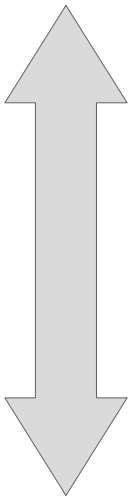
The Idea of a Programming Language

To give instructions to a computer this semester, we will use a **programming language**, in particular, the C programming language. The C programming language is composed of keywords that come from a natural language but are used without the ambiguity. Initially the C programming language syntax will look foreign to you as a beginner, but it is completely unreadable by the computer!

The computer has its own language, called a **machine language** that it can interpret as instructions specific to the individual machine that will perform the task. This machine language is written only in 0's and 1's because the internal circuits of a computer are made of switches, transistors, and other electronic devices can be in one of two states: off (0) or on (1).

It didn't take long for programmers to recognize that coding in a machine language is very challenging and the entire process of developing, debugging, and maintaining instructions in a machine language is not something most would enjoy doing. A process of evolution away from a machine language towards a natural language has been ongoing and has resulted in the programming languages of today.

Natural Languages



- **High-Level Languages**
- Use natural language elements that are easier to use, understand, and develop.
- Do not rely on specific hardware.

- **Low-Level Languages**
- Instructions written specifically for the hardware of one machine cannot be used easily on a different machine.
- Requires the programmer to consider the algorithm in small, incremental steps.
- Programmer left with the task of managing memory.

Machine Languages

How does the computer understand these high level programming languages if all a computer instructions are composed of 0's and 1's (binary numbers)?

Creating and Running Programs

To answer the previous question we must explore the process of turning a C program into a file that can be executed on the local machine.

The programming process is composed of steps which are repeated many times during development:

This copyrighted text licensed to on via

Writing and Editing the Program

The first step is **writing** the program (actually, the first step is developing your solution, but we are describing the implementation of the solution at this time) which will require the use of a piece of software known as a **text editor**. The text editor of choice for this semester will be **`vi`** (pronounced vee-eye). If you insist on another editor then you assume responsibility for understanding its operation and any potential compatibility issues with our server. We find the use of `vi` to be a powerful one for writing code. Please understand that there is a learning curve to `vi`, but with enough **practice** you too will be working with `vi` comfortably!

Saving the Changes

Please keep in mind the importance of **saving your work regularly**! Too many students in the past have lost a significant amount of unsaved work by a dropped connection, frozen computer, or power outage because they failed to save their work as they progressed.

- **You should test your work regularly too**! It is easy to write a lot of code and then only to find numerous errors or such a poor initial design that correcting all of the problems would be more difficult than starting over!
- **Testing regularly** should not imply you should create your solution while coding, but it cannot be stressed enough that when you write code it should be nothing more than the implementation of a well designed algorithm. Without planning, your solution is reduced to development through trial and error which tends to be an inefficient and frustrating process.
- **It is possible to make multiple-submissions for a given assignment.** However, it is only the last submission that is kept for grading. ALL previous submissions are over-written and cannot be restored.

In your collaborative lab groups, and for each lab assignment, you must designate a single individual who will make all submissions for the group.

Example: Assume we are implementing the solution to a problem and have progressed in our coding to the point that we want to test what we have input so far.

What should be done next?

1. Save the work to make sure the **source file** containing our C code has been stored and reflects our most recent changes.
2. Send the source file to the **compiler** so that the instructions within can be prepared for execution by the computer.

The Compiler

The compiler is in a category of software that is used to convert the source programming language code into the machine language of the computer.

Syntax is the set of rules of the programming language. The compiler will attempt to validate your source code with the syntax of the C programming language as it creates a new executable file.

Violating the rules of the syntax may result in an **error** in which no new executable file can be created.

This copyrighted text licensed to on via

The compiling process has one particular phase of interest to us:

- The **preprocessor** - reads the source code and prepares it for the translator. There are special instructions known as **preprocessor directives** that you can give the preprocessor to prepare the source code for translation.

On our server all of the steps of the compiling process are completed with a single instruction which leave many transparent to the programmer.

This copyrighted text licensed to

Process of Implementing Code

Program Execution (Testing/Running the Executable File)

on via

Upon successful compilation an executable file will be created. By default the name of this executable file is `a.out`. To execute, or run, the program you need only type `a.out` at the UNIX prompt. It is assumed that you are in the same directory as the `a.out` file while testing the program.

Each time you modify the source file you must recompile the code and create a new `a.out` to see the changes as reflected in your source code. A new executable file will replace any previously existing one. In most cases keeping the old executable file is not desired.

- If compilation of your program fails no new executable file will be created.
- A failed compilation attempt may not result an existing `a.out` file being removed/deleted.

Program Development

Our goal in CS 159: to be able to develop solutions in the form of logical algorithms as solutions to problems without considering any specific syntax of the programming language that you may potentially use.

Exposure to good programming practices is important to introduce the methods necessary to designing "a program that produces correct results, is well-designed, is efficient, and adheres to documentation and programming standards of the course."

Structure Charts

- Large programs are complex structures consisting of many interrelated parts and must be carefully designed.
- The structure chart shows how we are going to break out the larger problem into logical sub problems and how data will flow in between these modules.

Flowcharts

- The purpose of a flowchart is to describe visually the logic of an individual function (smaller task of the larger problem).
- A flowchart should be developed for each sub problem identified on a structure chart.

Each step invested in planning will assist you in eliminating logical errors that are often very time consuming to resolve later.

Problem: Given a loan amount, annual interest rate, and the term of the loan in years, determine the monthly payment.

Step #1 – What is specified of the program?

- Input – Three values. Loan amount, annual interest rate, and term. Output – Monthly payment.

Step #2 – Analyze the problem

- What type of mathematical operations do we need?

Variable	Description	Expression
`years`	Term of the loan in years.	Input by user
`numMonths`	Number of months for the loan.	`years * 12`
`annualRate`	Annual interest rate of the loan (as a percentage).	Input by user
`monthRate`	Monthly interest rate (decimal).	`(annualRate / 12) / 100`
`loanAmount`	Total amount to borrow.	Input by user
`P`		`pow((1 + monthRate), numMonths)`
`Q`		`(P / (P - 1))`
`monthPayment`	Monthly payment.	`loanAmount * monthRate * Q`

This copyrighted text licensed to on via

Step #3 – Designing a Software Solution

- It is here that we consider individual steps, or instructions, that we must implement to solve the problem.

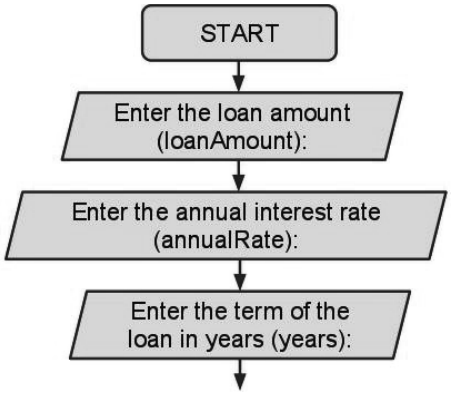
Flow Chart Symbols/Types of Instructions

Symbol	Description	Symbol	Description
	Terminal - Start and End of algorithm **Course Standard:** For this course, only one starting and ending point (terminal) is permitted. All other implementations violate standards.		Conditional Statements • Always label your true and false paths!
	Assignment Statements / Expressions • Includes function calls!		Repetition Constructs • Always label your true and false paths!
	Designate Data for Input and/or Output • Don't worry about width modifiers and conversion codes.		Connector • Because paper isn't infinite in any dimension.

This copyrighted text licensed to on via

This copyrighted text licensed to on via

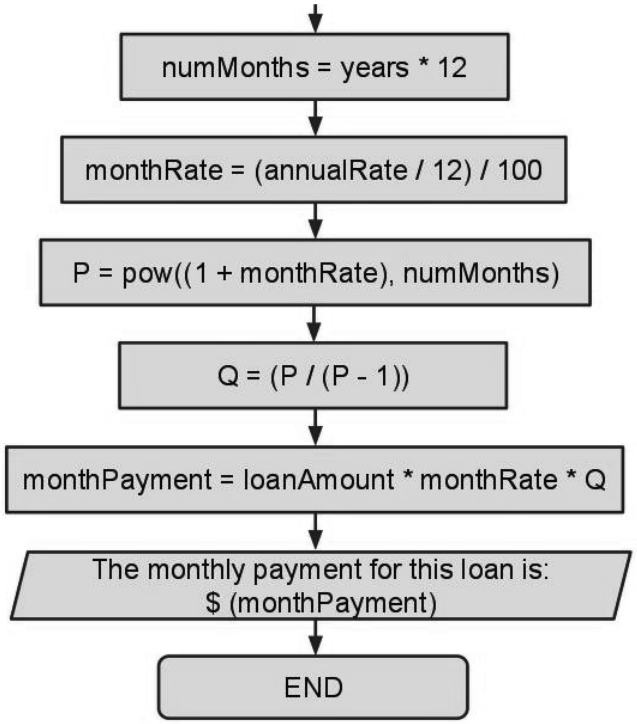
Dealing with the input first:


START
Enter the loan amount (loanAmount):
Enter the annual interest rate (annualRate):
Enter the term of the loan in years (years):

You will notice in the C text that a descriptive prompt is not present when accepting (reading) input. It is a good idea to at least reference the name of the variable in the I/O symbol if it is not obvious.

- Flow charting is discussed in Appendix C of the C text. The symbols used are potentially applicable to any programming language.

Continue with the calculations and conclude with output:


numMonths = years * 12
monthRate = (annualRate / 12) / 100
P = pow((1 + monthRate), numMonths)
Q = (P / (P - 1))
monthPayment = loanAmount * monthRate * Q
The monthly payment for this loan is: $ (monthPayment)
END

Continue with writing, saving, compiling, testing, revising…

- **Testing**. In addition to the example executions provided on all assignments you, or your group for lab assignments, should consider creating additional test cases that verify the correctness and efficiency of your solution.
- **Refinement** is an important, but often neglected, step to the problem solving process. Your first solution isn't always the best and further refinement of the algorithm might improve the efficiency of the program and eliminate any previously unobserved errors.

The Structure of a C Program

	Special instructions to the preprocessor that tell it how to prepare your program for compilation.
	Declaration of "objects" that will be visible (usable) to all parts of the program. **Course Standard:** Variables will **NEVER** be acceptable as global.
	The final section represent the functions. A C program is made up of one or more functions, one of which must be named `main`. **Course Standard:** The two sections of a function **MUST** not overlap! Additional functions will be introduced in chapter 4.

This copyrighted text licensed to on via

Our First Program

```
#include<stdio.h>

int main(void)
{
  printf("Hello World!\n");

  return(0);
}
```

Do not confuse <stdio.h> with <studio.h>

Why is the void in this program optional?

Dissection of above program

Preprocessor Directives Section

- The only preprocessor directive statement is #include<stdio.h>
- This particular statement requests that a standard library (stdio.h) be included in order to permit the use of the standard input/output functions (such as printf) within the program.
- The stdio.h library will appear in all C programs you write this semester. Other libraries, such as math.h and stdlib.h will be introduced later this semester. See Appendix F of your C programming text for a list of libraries and available functions.

Global Declarations Section

- There are no global declarations in this program.
- There will **NEVER** be global **variable** declarations in your program.
- The main function does not require a declaration and since there are no other user-defined functions, this section remains empty.

Always Remember: The **declaration section** and **execution sections** MUST NEVER overlap! You **must** complete all declarations prior to the beginning of executable statements.

Function Section

- There is only one function in this program, that function is named main.
- The main function is where execution will begin (and end) in a program.
- Every function has two distinct sections:

1. Local Definition Section – establishes data needs for current function.
2. Statement (Execution) Section – logic to accomplish the task of the function.

This copyrighted text licensed to on via

There are only two executable statements in the `main` function. The first statement will print the message in quotes on the monitor.

- One special character `\n` included inside of the quotes will print a new line (similar to hitting the enter key on your keyboard).
- Another special character (not used in this example) `\t` is used to display a tab, but because tabs are interpreted differently it is **best to avoid** their use for a consistent appearance of your output.

The second statement is `return(0)`, this statement indicates that control of the program returns from the `main` function to the operating system, terminating the program. Later this semester we will see how the `return` statement is used to return control of the program, with or without a value, from one user-defined function to another.

This copyrighted text licensed to

More Observations from Our First Program

`#include<stdio.h>`
 	
`int main(void)`
`{`
`  printf("Hello World!\n");`
 	
`  return(0);`
`}`
	Course Standard: All statements between the `{` and `}` are indented two spaces.

1. Notice the use of `{` and `}` that indicate the beginning `{` and end `}` of the `main` function. Any C structure that has a body, the contents between the curly braces, will not terminate with a semicolon.
2. Most other C statements (and declarations) will terminate with a semicolon. The rules above usually hold true but there will always be exceptions!

on via

Commenting: It is reasonable to expect that a good programmer would be able to read code written by another, but sometimes the meaning of the code (and/or the thought-process of the coder) is not obvious in a logically complex segment. It is helpful if the person who wrote the code adds detailed notes to assist the reader who might be a grader, tutor, fellow student, or a colleague. **Comments** are internal documentation which are ignored by the compiler and are intended to be read by another viewer of the source code.

The C programming language, any many other languages, provides two methods of commenting:

1. Single-line comments
2. Multiple-line comments

Single-line comments only span from the start of the comment to the end of the current line. A single-line comment begins with `//` and the text that follows is ignored for the remainder of the current line.

```
exam_average = exam_sum / numberStudents;    //CALCULATING THE EXAM AVERAGE
```

Multiple-line comments are used when your comments span more than one line. Instead of starting every line of code with `//`, we use `/*` to represent the start of the comment and `*/` to signify the end of the comment. The headers you will be required to use in all of your programs this semester are examples of a multiple-line comment. **All text** between the `/*` and `*/` is ignored by the compiler.

```
/*****+--*--****-*-----*-****-*--*--*-***-*---*-*-*************************
*
*  Lab #: SPECIFY ASSIGNMENT NUMBER
*
*  Academic Integrity Statement:
*
*  We have not used source code obtained from any other unauthorized source,
*  either modified or unmodified. Neither have we provided access to our code
*  to another. The effort we are submitting is our own original work. We have
*  not made use of any AI generated code in this solution.
*
*  Program Description:  DESCRIBE YOUR ALGORITHM.  WHAT DOES YOUR PROGRAM DO?
*                        AND HOW DOES IT DO IT?
*
******+--*--****-*-----*-****-*--*--*-***-*---*-*-*************************/
```

Rules regarding comments:

- Comments can appear anywhere in the program.
- Don't comment EVERY line of code you write.
- You must include an assignment header in every assignment you submit. A well-written description in the header will reduce the need for comments within the code.
- You **MUST** comment EVERY variable in your program.

Note: Multiple-line comments cannot be nested.

Identifiers allow us to name data and other objects in a program.

- Data used by a program may be stored by the computer in a unique memory address and as programmers we only need to recall the identifiers we select rather than specific memory addresses to store and retrieve data values from the memory of the computer.

Good Identifier	Bad Identifier
`numStudents`	`x`
`depth`	`DEPTH`
`examAvg`	`exam-avg`
`first_lab`	`1st_lab`

Documentation Standard:

- The name of an identifier should be related to its purpose in the program.
- Rarely are single character identifiers meaningful.
- Identifiers in all caps represent a symbolic/defined constant, that is, a value that will not change during the execution of the program.

This copyrighted text licensed to on via

Rules regarding the selection of an identifier:

- The first character must be an alphabetic character.
- Must consist only of alphabetic characters, digits, or underscores.
- Only the first 63 characters of an identifier are significant.
- Cannot duplicate a reserved word. (Words used in the C statements include; `main`, `int`, `return`, `void`, see Appendix B)

Types (of Data)

What is determined by a data type?

- The amount of memory necessary to store a value of a given type.
- How a value of a given type is stored in the memory of the computer.
- Which operations can be performed on the value. Some operations are restricted to particular data types.

An **integer** (`int`) type is a number without a fraction part. C supports different sizes of the integer data type: `short, int, long,` and `long long.`

- The type of integer you will use depends on the potential range of integer data you plan to store. The longer types will require more memory.

`int` range, `-2147483648` to `2147483647`	Actual ranges will be system dependent.

A **character** (`char`) is any value that can be represented in the "alphabet" of the computer. Each character has a corresponding small integer value. This standard table of integer values alongside the equivalent characters is known as the **ASCII table**. You will find the ASCII table in Appendix A of the text.

- Since a character is stored in memory as an integer representing the ASCII code of that character it is possible to perform arithmetic operations on characters!

Boolean (`bool`) values represent logical data (either true or false). Prior to the C99 standard of the C language only integers were used to represent truth values. The compiler currently used by the course does not recognize the `bool` data type by default.

- For backward compatibility, nonzero numbers represent true and a zero value can be used to represent false.
- **We will not use the `bool` data type this semester**, an integer value can be used in its place.

This copyrighted text licensed to on via

A **floating point** (`float`) number has a fractional part, such as `3.14159`. C supports different sizes (smallest to largest) of the floating point data type: `float, double,` and `long double`. The most common implementations will be `double` and `float`. The longer types will provide additional precision and require more memory.

> **Note:** Many programming references will refer to any numeric value that is not an integer as a real number. This usage may not be consistent with what you know about real and imaginary numbers from mathematics.

This copyrighted text licensed to

Variables

A **variable** is a named memory location that has a data type.

Each variable in your program must be declared and defined. In C, a declaration is used to name an object (such as a variable) and definitions are used to create the object.

- Declaration - gives the variable a name.
- Definition - reserves memory for the variable.

To create a variable (doing both declaration and definition in one step), you first need to list the intended data type of the variable, followed by the selected identifier, and a terminal semicolon.

- `int count; //COUNTS THE NUMBER OF GPAs GREATER THAN INPUT VALUE`

You need to know: A variable is defined to represent a particular data type and can only store and represent values of that type!

> **Course Standard:** Declare only a single variable per line.

on

Those with previous programming experience in languages similar to C may know that it is possible to declare multiple variables in one statement. However, the course **programming standards** do not permit this practice. Organizing the data needs of a function can be a challenge and a single declaration per line assists with this and improves the documentation and readability of code.

via

Variable Initialization

It is possible to **initialize** a variable at the same time it is created.

```
int count = 0;
```

The above will initialize the space in memory reserved for the integer `count` to the value of zero. We cannot assume the memory assigned to a variable has any default value unless we initialize the variable. Uninitialized variables will be assigned a location in memory that may have existing data from a prior use. Such data is referred to as **garbage** because it has no predictable value for the programmer.

> **Course Standard:** Move complex initialization expressions to the executable statement section of a function.

Constants - Literal Constants

A **literal constant** is an unnamed value used to specify data. For example, we want to calculate the circumference of a circle:

```
circumference = 2 * 3.14159 * radius;
```

- The values `2` and `3.14159` are both literal constants in the above expression. We may often refer to such values as being "**hard coded**" when they are not represented by a symbol (next topic).

Constants - Symbolic/Defined Constants

The `#define` preprocessor directive is used to create **symbolic constants**.

When this statement appears among the preprocessor directives, all subsequent occurrences of `symbol` will be replaced by `replacement value` automatically before the program is compiled.

Note: The replacement of symbols for the values they represent will NOT take place inside of quotes.

Why use **symbolic/defined** constants?

- It is difficult to determine what a literal constant value represents in an arithmetic expression when the program is long and includes many complex calculations. **The use of constants helps document the program by giving meaning to operands in an expression.**
- Another great reason to use constants is to make your program more **maintainable.** For example, perhaps there are 100 students in a course. You might use the value 100 when loading data or determining the average, however, you might also use 100 to convert .853 to 85.3%. Now, what if the number of students changes to 105? You then need to search your program to determine which of those 100 values that occur in the program are related to the number of students in the course and change only those to 105.

Course Standard: Variables declared with the `const` type qualifier are not permitted. The C programming text may refer to these variables as memory constants and their application is outside any assignment presented in the course.

This copyrighted text licensed to on via

Symbolic Constant Examples:

Preprocessor Directive	Good or Bad Usage? Why?
`#define NUMSTUDENTS 100`	Good, clearly represents the number of students in the course.
`#define ONE 1`	Bad, vague! If the value `1` has several purposes in your program then create a constant for each!
`#define NUMBEROFEXAMS 4;`	Bad. Semi-colons are not typically used. A constant will substitute **all** values that follow the name.
`#define MaxPtsOnExam 100`	Bad, always use all upper-case letters in a symbolic constant name.
Course Standard: Attempt to minimize your use of literal constants and maximize your use of symbolic constants.	
Comparing Symbolic/Defined Constants with Variables:	

Misapplication of Symbolic/Defined Constants

What is in your code:	What is really being compiled on line #10:

`3 #define APPROX_PI = 3.1415;`
`4`
`5 int main(void)`
`6 {`
`7  float radius = 4.25;`
`8  float circumference;`
`9`
`10  circumference = 2 * APPROX_PI * radius;`
	`circumference = 2 * = 3.1415; * radius;`
	
	• The substitution of all that follows the symbol `APPROX_PI` generates an error on line #10.
	• When observing an error on a line that contains a symbolic/defined constant remember that the source of the error may be in the definition at the top of your program.

This copyrighted text licensed to on via

Formatted I/O (Input and Output)

The `printf` function is used to display messages and formatted data to the monitor.

- To print data the `printf` function needs two things: instructions for formatting the data and the actual data to be printed.

We saw the following in an earlier example to display text only:

- `printf("Hello World!\n");`

It is possible to print the data represented by constants or variables in a `printf` statement:

- `printf("The year is: %d\n", 2026);`

Where `%d` is a **placeholder** for the integer literal constant 2026.

- **Reminder:** The term **placeholder** is used here in the notes, the book will refer to the `%` as a start token to be followed by a conversion code (in the example above a "`d`" is the conversion code for an integer value).

The format of a `printf` statement is: `printf( format string, data list );`

- Where the **format string** is everything inside of the double-quotes and the variables to be printed are after the comma in the **data list**.
- A data list may contain multiple values that are separated by commas.

You can **print all of the typical C data types with** `printf` by using different conversion codes within placeholders:

- `int` (integer family values) uses **%d**
- `float` (floating-point family values) uses **%f**
- `char` (single character values) uses **%c**

This copyrighted text licensed to on via

A more comprehensive list of conversion codes:

Family	Type	Size	Code	Example
Integer	`char`	None	`c`	`%c`
	`int`	None	`d`	`%d`
	`long`		`d`	
	`long long`		`d`	
Floating-point	`float`	None	`f`	`%f`
	`double`		`f`	
	`long double`		`f`	

Width Modifiers:

- A width modifier is used with a conversion character to reserve a given amount of space for a value to be displayed.
- If the number of characters in the data to display is less than that specified by the width modifier then the data will be right aligned in the spaces reserved.
- Left alignment can be achieved by prepending the − prior to the width modifier. This option is known as the flag.

Precision Modifiers:

- A precision modifier is used with floating-point data to determine the number of digits to display to the right of the decimal point.

Placeholder Summary:

%				**Size**	**Code**

The flag attribute is the character used for subtraction and will shift the default alignment from right to left. For example: `%-10d`

This copyrighted text licensed to on via

Example of width modifiers, precision modifiers, and variable precision modifiers:

```
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
Circumference:                 26.70
Circumference:            26.703514
Circumference:               26.704
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
```

Notes on the code segment:

- The desired number of decimal places is stored in a variable named `decimalPlaces`.
- This value is then used to represent the desired number of decimal places to display and replaces the `*` in the placeholder.
- The width modifier is a fixed value of 18. This, too, could be represented by a variable similar to how it was accomplished for the precision.

Code used to generate the output on the left:

```
#include<stdio.h>

#define APPROX_PI 3.14159

int main(void)
{
  float radius = 4.25;
  float circumference;

  int decimalPlaces = 3;

  circumference = 2 * APPROX_PI * radius;

  printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
  printf("Circumference: %18.2f\n", circumference);
  printf("Circumference: %18f\n", circumference);
  printf("Circumference: %18.*f\n", decimalPlaces, circumference);
  printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");

  return(0);
}
```

The primary input function we will use this semester is called `scanf`

The format of the `scanf` function is similar to that of `printf` in that it has two sections separated by a comma.

```
scanf( format string, address list );
```

- The **format string** will list the types of data expected as input. It is possible that you will read in more than one data in a single `scanf` statement. The **address list** indicates where in the memory of the computer the input value should be stored.

```
printf("Enter current year -> ");
scanf("%d", &year);
```

The user will enter an integer that will be assigned to the variable `year`.

- What about the `&` before `year`?
- The `&` is the **address operator** and refers to the address represented by the variable `year`.
- The value entered by the user will be stored in memory at the address represented by `year`.

This copyrighted text licensed to on via

The `scanf` function uses the same placeholders as `printf`:

- `int` uses **%d**
- `float` uses **%f**
- `char` uses **%c**

Remember: The `\n` represents a new line character for output and would not be anticipated in a `scanf` statement.

This copyrighted text licensed to

Error Discussion

We will not be able to cover every specific error you may encounter during the term, but we will place errors in various categories today and try to mention common errors as we work our way through new topics. The goal of this short section is to help you better understand the error you are trying to resolve and to communicate that with another who may be attempting to provide assistance.

Classification of errors:	Try to answer the following questions whenever you encounter an error:
1. Syntax error 2. Logical error	1. When does the error arise? 2. What did you observe about the error?

The syntax error

You will be made aware of a syntax error when you attempt to compile your C code. It may also be said that the syntax error is a **compile-time error**. Typically, this error arises due to a mistake you made that violates the **syntax rules** of the C programming language causing the compiler to stop its work.

- Common syntax errors include misplacing a semicolon, failing to pair all quotes, and other typos.
- **A warning** is feedback provided by the compiler that you have potentially made a mistake. All warnings should be remedied before you submit an assignment. If you have set up your account correctly you will be compiling utilizing the warnings option. Without this option active, or by making use of an alternative compiler, you may not be made aware of any warnings.

on via

Creation of Executable Files

A new executable file (`a.out`) will not be created when the compiler must terminate its work due to a syntax error. The keyword here is **error** and it is commonly included as part of the messages created by the compiler. If the compiler reports **only warnings** then it is likely that a new executable file (`a.out`) will be created. When issuing a warning the compiler has detected a potential misuse of the syntax which will often appear as a logical error when the program executes.

Example Warning Message:

```
test.c: In function 'main':
test.c:10: warning: 'ct' is used uninitialized in this function
```

The run-time errors

This is any error that occurs when the program is running. The run-time error may result in a crash. A crash is an event that causes the execution of the program to cease unexpectedly. Two very common run-time errors that result in a crash are the **segmentation fault** and the **floating exception**. Why aren't these errors detected by the compiler? The compiler may not know an attempt to divide by zero (floating exception) will occur until after the code has been compiled and the user enters a zero for a value that will then be used in the denominator of an expression.

The more difficult error to diagnose is a run-time error that does not lead to a crash! Such an error is often noticed because the output (results) is not as expected.

Logical error example, determine the average of three numbers:

```
average = x + y + z / 3;
average = (x + y + z) / 3;
```

Both statements would be syntactically correct; the first statement will not cause a crash, but will result in the display of an incorrect value. The second statement is the desired result without the logical error.

- A **logical error** is a run-time error in which **the programmer is responsible** for implementing a flaw in the logic of the program.

Debugging

- The process of resolving errors is known as debugging. All semester we will look at various approaches to debugging your errors including:
  - Inserting **diagnostic print statements** into your code to verify the expected values at a given point in time in the execution of your program.
  - Writing small sections of code that accomplish a sub-task within the larger problem that you are trying to solve.
  - Developing test cases that represent a variety of expected input.

Chapter 3 – Structure of a C Program

Expressions

An expression always reduces to a single value in the C programming language.

Expression Terminology Review:

- **operator** - "language-specific syntactical token that requires an action to be taken." For example: `+, -, *, /,` and some others that may be new to you. If you can take one thing away from the definition above, focus on "language-specific" in that in the C programming language the operators and their meanings might be different from what you may have seen in mathematics or another programming language (namely MATLAB, Python, Excel).
- **operand** - "receives an operator's action." In other words values operated on by the operator. Depending on the operator there may be one, two, or more operands necessary to evaluate the operation.

This copyrighted text licensed to on via

Another idea that you should be familiar with in some form is the idea of operator **precedence**. The precedence of an operator determines (1) to which operator an operand belongs and (2) the order in which it will be evaluated in an expression. The inside cover of the text book has a list of the **order of precedence** for the C language.

Printing Expression Evaluations:

```
int a = 17;  //FIRST OPERAND
int b = 5;   //SECOND OPERAND

printf("%d + %d = %d\n", a, b, a + b);

printf("%d / %d = %d\n", a, b, a / b);

printf("%d %% %d = %d\n", a, b, a % b);
```

Output generated:

Why are there two `%` characters in the format string of the third `printf`?

- Because the `%` already has a special meaning in the format string of a `printf` it takes two to display one to the screen.

Note: Both operands of the modulus operator must be integral types.

Assignment Expressions

An assignment expression evaluates the operand on the right side of the assignment operator (=) and saves that value in the memory of the variable on the left.

Note: The left operand in an assignment expression must be a single variable.

Examples of **Simple Assignment**

```
int x;
int y = 2;

x = 3 * 4;
x = y * 3;
x = x + y;
```

Course Standard: Place a space between your operators and operands!

Examples of **Compound Assignment**

- It is common to add (or apply another common mathematical operation to) a value to a variable and store that new value back to the variable.

This copyrighted text licensed to on via

For example, `x = x + 3;`

- Consider what is happening above in terms of memory. On the right we retrieve the value of `x` from memory and add three to it and then store the sum back in to the memory location represented by `x`.

The compound assignment example:

- `x += 3;  //SAME AS x = x + 3;`

Compound Assignment Expression	Equivalent Simple Assignment Expression
`x += expression`	
`x -= expression`	
`x *= expression`	
`x /= expression`	
`x %= expression`	

Example:

```
int x = 3;
int y = 5;
int z = 7;
float a = 2.5;
float b = 5.0;

x += 4;
y /= 2;
a *= 2;
b /= 2;

z %= 4;

printf("x = %d y = %d z = %d\n", x, y, z);

printf("a = %.2f b = %.2f\n", a, b);
```

This copyrighted text licensed to on via

Example #2:

```
int x = 3;
int y = 5;

x /= x + y;

printf("x = %d y = %d\n", x, y);
```

Is the result `x = x / x + y` **or** `x = x / (x + y)`?

- Answer: In the case above, the expression will be interpreted as: `x = x / (x + y)`

What does this tell you about the operator precedence of `/=` and `+` ?

- The addition operator has precedence over the compound assignment operations.

Prefix & Post-fix Operators

- Within many programs there is a need to count and this is often done in increments of +1 or -1. These two notations, prefix and post-fix, provide a shorter alternative to the simple or compound equivalents but can also be combined with other expressions and it is there where we find that the position of the operators with respect to the operand makes a difference.

A **post-fix expression** involves one operand followed by the operator.

```
int x = 0;

x++;  //POSTFIX INCREMENT
printf("The value of x is: %d\n", x);

x--;  //POSTFIX DECREMENT
printf("The value of x is %d\n", x);
```

A **prefix expression** involves the operator followed by the operand.

```
int x = 0;

++x;  //PREFIX INCREMENT
printf("The value of x is: %d\n", x);

--x;  //PREFIX DECREMENT
printf("The value of x is %d\n", x);
```

This copyrighted text licensed to on via

Why are there no recognizable differences between the post-fix and prefix examples above?

- When used on a line without any other expression the post-fix and prefix operations are interchangeable.
- Additionally, under those same conditions a simple or complex assignment could be substituted for the post-fix or prefix operations.

When are the differences between the post-fix and prefix operations demonstrable?

- When those operators are used within another expression.

Result of Post-fix:	**Result of Prefix:**

Example to demonstrate the differences between the prefix and post-fix operations:

```
int x = 0;

printf("The value of ++x is: %d\n", ++x);

x = 0;  //RESET x

printf("The value of x++ is: %d\n", x++);

printf("final value of x: %d\n", x);
```

This copyrighted text licensed to on via

Multiple Modifications in a Single Expression

```
int x = 0;
int y = 0;

y = x++ + ++x - --x - x--;

printf("x: %d y: %d\n", x, y);
```

```
int x = 0;
int y = 0;
int z = 0;

x = z++ + ++y;

printf("x: %d y: %d z: %d\n", x, y, z);
```

An expression such as `y = x++ + ++x - --x - x--` may not be interpreted the same way by different compilers. In general, undefined operations should not remain in your program even if you have determined how exactly the operation is defined for our server.

Warning: In the C programming language, if a single variable is modified more than once in an expression, the result in **undefined**.

Mixed and Single Data Type Expressions

We have seen examples already of expression evaluations that produce results other than what we expect if we were in mathematics (or using a calculator, Excel, or MATLAB).

```
3 / 2 = 1
```

- The above example is another one of those situations! What happens here in C programming is that the division of two integer values must result in another integer!

However, if we mix the types as seen the statement below the remainder is preserved in the sum.

```
3 + 2.1 = 5.1
```

The **mixed type expression** in the previous example will **implicitly type convert** the integer 3 to a floating-point 3.0 such that the expression is evaluated as if two floating-point values were present. Do you see any problem converting 3 to 3.0 before it is added to 2.1?

- No, there is no potential loss of data and the implicit conversion is referred to as **safe**.

This copyrighted text licensed to on via

What does it mean to be implicit? The computer does it without the programmer explicitly asking for the conversion to be done! The rules for implicit data type conversions are illustrated in the image above.

- Are there explicit data type conversions and, if so, how is an **explicit type conversion** implemented?

Example requiring an explicit type conversion: I want to calculate `3 / 2`, but I need the answer to be `1.5`!

We saw in the mixed type expression that the integer was promoted on the data type hierarchy to a `float` implicitly. We can request a type conversion **explicitly** on one of the two operands in the division expression forcing the division to involve a floating-point and an integer. The quotient will then be generated as a floating-point value because the expression is now a mixed type expression and the second operand (the denominator in the example below) is changed implicitly.

```
(float) 3 / 2 = 1.5
```

Try the following:

Expression	Evaluation
`3 / (float) 2`	
`(float) 3 / (float) 2`	
`(float) (3 / 2)`	

What do the results of the expressions above say about the precedence of the explicit type conversion?

Another Math Review: Rounding of numbers is a common task in problem solving. You must keep in mind the rules regarding rounding as we program this semester or better yet be able to conduct various tests to identify how rounding is taking place with your values.

- Sometimes an "extreme" form of rounding takes place called **truncation**. Truncation is removing of the entire decimal value and giving no consideration to rounding the integer that remains.
- The `floor` function (found in `math.h`, but also a commonly used function in MATLAB and Excel) will remove a remainder and reduce the number to the next smaller integer value. Note how its behavior is similar to that of the conversion of `float` to `int`.

Expression	Evaluation	Expression	Evaluation
`floor(6.99)`		`(int)6.99`	
`floor(6.5)`		`(int)6.5`	
`floor(6.0)`		`(int)6.0`	
`floor(6.75 + 0.5)`		`(int)(6.75 + 0.5)`	

This copyrighted text licensed to on via

Assignment Conversions:

- This type of conversion occurs when the data type of the expression on the right-hand side of an assignment expression does not match the data type of the variable on the left-hand side of the assignment operator.
- The result of the expression on the right-hand side of the assignment expression will be converted to match that of the variable on the left-hand side. Such a **conversion has the potential to be unsafe.**

This copyrighted text licensed to

Example - rounding and truncation

```
int a = 3;
int c = 1;
float x = 3.99;
float y = 2.29;
float z = 8;

printf("The value of z is %f\n", z);

c = x; //ASSIGNING A FLOAT TO AN INT - ASSIGNMENT CONVERSION

printf("The value of c is %d\n", c);

printf("The value of x is %.1f and y is %.1f\n", x, y);

printf("Using mixed up placeholders: \n");
printf("x is %d and a is %f\n", x, a);

printf("Using a cast to fix placeholders: \n");
printf("x is %d and a is %f\n", (int)x, (float)a);
```

on via

Note: The warning issued as a result of incorrect conversion codes is an indication of another **undefined** behavior. How the values displayed are determined is based on the individual processor.

Reminder: An explicit type conversion applied to a variable WILL NOT alter the variable's data type for the remainder of a program. You can consider the type conversion is applied AFTER the data value for a variable has been retrieved from its allocated memory.

How the value of variables are changed:

- The assignment operator has the ability to alter the value of a variable. Similarly, the compound assignment operations can change what is stored in the memory assigned to a variable.
- The use of the address operator in a `scanf` function call will change what a variable is storing in memory.
- Prefix and post-fix increment and decrement will make a change to the value of a variable. The particular operator used will determine **when** this change will take place.

This copyrighted text licensed to on via

Selection via Calculation

Wait, this is not the chapter on selection, is it? Where are the `if/else if/else` constructs? How can we simulate selection with a calculation? The fundamental idea here is that we can create an expression that can be activated or not based on the value of a factor which is multiplied by the expression to determine whether it remains a part of the final calculation. The factor is often a result of comparing two values to determine which of the two is larger. Given that this chapter has placed an emphasis on data types it is indeed that idea which plays a major part in the calculation of such a factor.

Example: Given two integers display the larger of the two.

```
 1 #include<stdio.h>
 2
 3 int main(void)
 4 {
 5   int x;
 6   int y;
 7   int max;
 8
 9   printf("Enter two integers: ");
10   scanf("%d %d", &x, &y);
11
12   max = x / y * x + y / x * y;
13
14   printf("Max value input is: %d\n", max);
15
16   return(0);
17 }
```

Example Executions:

```
Enter two integers: 5 8
Max value input is: 8

Enter two integers: 9 6
Max value input is: 9

Enter two integers: 10 3
Max value input is: 30
```

- See the problem when one of the numbers is two or more times greater than the other?
- How do we fix this? How can we make sure that the quotients calculated are always 0 or 1?

Revision to eliminate multiplying the larger value by a factor greater than one:

```
factor1 = x / y;
factor2 = y / x;

factor1 = (factor1 + 2) % (factor1 + 1);
factor2 = (factor2 + 2) % (factor2 + 1);

max = factor1 * x + factor2 * y;
```

- Two integers `factor1` and `factor2` are introduced to store the values comparing `x` and `y`.
- We know the value that is zero represents the smaller of the two values and the value greater than zero is the larger.
- The modulus operator is used here to scale all non-zero values to 1 while not changing the factor storing zero because `2 % 1` is zero.

Example Executions of Revised Program:

```
Enter two integers: 10 3
Max value input is: 10
```

```
Enter two integers: 8 8
Max value input is: 16
```

- We still have a problem when both integer values are the same!

Revision to eliminate the duplication of the output when the two values entered are the same:

```
factor1 = x / y;
factor2 = y / x;

factor1 = (factor1 + 2) % (factor1 + 1);
factor2 = (factor2 + 2) % (factor2 + 1);

max = factor1 * x + factor2 * y;

max = max / (factor1 + factor2);
```

- Only when the sum of the factors is two are both values the same. In all other cases the sum of the factors will be one.
- With that being the case we only need to divide the `max` by the sum of the factors to remove this unwanted doubling of the value.

Example: Using what we now know about data types and expressions, given the total number of points earned during a semester calculate the letter grade earned.

```
Enter total points earned: 470
Points Earned: 470  Grade Earned: A

Enter total points earned: 415
Points Earned: 415  Grade Earned: B
```

```
Enter total points earned: 305
Points Earned: 305  Grade Earned: D

Enter total points earned: 304
Points Earned: 304  Grade Earned: F
```

Solution:

```
#define A_MIN 470
#define B_MIN 415
#define C_MIN 360
#define D_MIN 305

  int totalPoints;   //TOTAL POINTS EARNED BY STUDENT
  char courseGrade;  //GRADE EARNED BY STUDENT
  int gradePoints;   //4 - A, 3 - B, 2 - C, 1 - D, 0 - F

  printf("Enter total points earned: ");
  scanf("%d", &totalPoints);

  gradePoints = totalPoints / A_MIN;
  gradePoints += totalPoints / B_MIN;
  gradePoints += totalPoints / C_MIN;
  gradePoints += totalPoints / D_MIN;

  courseGrade = 'F' - gradePoints;
  courseGrade -= totalPoints / D_MIN; //TO ACCOUNT FOR 'E'
  printf("Points Earned: %d  Grade Earned: %c\n", totalPoints, courseGrade);
```

This copyrighted text licensed to on via

Why use functions?

This copyrighted text licensed to on via

Top-Down Design

The identification of smaller tasks within the larger problem is known as **factoring** and results in small manageable sub-parts of a program to be designed and individually developed then implemented. This process of factoring tasks into more manageable parts is known as **top-down design**.

When does the factoring stop? This factoring should continue until the task "consists only of elementary processes that are intrinsically understood and cannot be further subdivided." (page 150, C Programming text)

- In simple terms, a function should consist of a **single task**. Be careful not to divide tasks such that the resulting functions are trivial. For example, the act of input of a value includes the `printf`, `scanf`, and validation of the input. To divide this task of input further would result in functions that are trivial.
- More good examples of "single task" functions can be found in the `math.h` library. These functions accept as input one or more values and generate a single value as a result. No additional tasks such as input, output, or further calculations are included alongside the task of these functions. The "single task" philosophy does improve the likelihood that the function can be reused in the future.

When is a new function necessary?

- A function that accomplishes only a single task is known to be **functionally cohesive.** Our goal is to develop such functions and to avoid placing multiple unrelated tasks inside of the same function.
- The logic behind the task that a function accomplishes in a program should **appear only once** in the program. The motivation for this idea is that we desire to **eliminate redundant code**, that is logic which appears in more than one place in a program.
- The task that the function is completing should be **testable** by itself separate of the rest of the program.

This copyrighted text licensed to on via

Basic Function Terminology

Behind ALL WELL-WRITTEN programs is the same fundamental component, the function. The function will be the implementation and contain the solution of the single sub-task previously described. What is a function? According to the Gilberg and Forouzan text, "A named block of code that performs a process within a program; an executable unit of code, consisting of a header, function name, and a body, that is designed to perform a task within the program."

- When the services of a function are requested it must be **called**. The function making the call is known as the **calling function** and the function being called is the **called function**.
- It is common that data needs to be exchanged between the calling and called functions.
  - The act of sending (passing) data to a function is known as **parameter passing.**
  - It is possible that the function may complete a calculation and wish to share the result with the calling function. We refer to this as **returning** a value from a function.
  - In the C programming language, a function is only capable of **returning at most one value.**

```
#include<stdio.h>

void greeting(void); //FX DECLARATION

int main(void)
{
  //LOCAL DECLARATIONS - NONE

  //EXECUTABLE STATEMENTS

  greeting(); //FX CALL

  return(0);
}
```

```
//FUNCTION DEFINITION

void greeting(void)
{
  //LOCAL DECLARATIONS - NONE

  //EXECUTABLE STATEMENTS

  printf("Hello World!\n");

  return;
}
```

In the above example we observe a `main` function that contains two statements, one statement is a **call** to a function named `greeting` and the second statement is the terminal return [`return(0);`] statement. When the function `greeting` is called by `main` control of the program leaves `main`. The function `greeting` will execute its two statements which will print `Hello World!` to the monitor and then return control of the program to the calling function `main`. The `main` function will continue from where it left off following the function call, in this case, `main`'s next statement will be the terminal `return(0)` statement.

- The **sequence** of which **instructions are executed** in the program now jumps from `main` to `greeting` when the function is called.
- It is "remembered" where in `main` the function call was made so that when the called function (`greeting`) terminates the execution of statements in `main` will continue with the statement following the function call.

User-Defined Functions

As with variables, functions must be both **declared** and **defined**.

The **function declaration** will appear in the global declaration section of the program.
The function declaration statement will tell the compiler:
(1) The type of value being returned from the function.
(2) The name of the function.
(3) The data being sent to the function. The number of parameters, type of parameters, and order of parameters is determined by the declaration of the function.
The **function definition** contains the code needed to complete the task.
	What is the compiler looking for?
	• All three components of a user-defined function, (declaration, definition, and call) are in agreement.
	What is the error generated when the definition cannot be found?
	`undefined reference`
	What is the error generated when the declaration cannot be found?
	`warning: implicit declaration of function`

Basic Function Designs		**Yes Parameters**	**No Parameters**
1. A function with no parameters that returns no value. 2. A function with no parameters that returns a value. 3. A function with parameters that returns a value. 4. A function with parameters that returns no value. **What is "void" about a `void` function?** • Such a function does NOT return a value.	**Yes Value Returned**		
	No Value Returned		

What does it mean to describe a function by a data type?

- Describing a function by a data type is a reference to the type of value that it returns.

This copyrighted text licensed to on via

A Function with no Parameters that Returns no Value

- The `greeting` function is an example of a `void` function and are useful to handle tasks that require no value to be returned.

```
#include<stdio.h>

void welcome(void);  //FUNCTION DECLARATION

int main(void)
{
  welcome();  //FUNCTION CALL

  return(0);
}

/*****+-**--***-*---*-*-********---**-*****--*********************************
*
*  Function Information
*
*  Name of Function: welcome
*
*  Function Return Type: void
*
*  Parameters (list data type, name, and comment one per line): NONE
*
*  Function Description:  Displays the welcome message to the user.
*
******+-**--***-*---*-*-********---**-*****--*********************************/
void welcome(void)
{
  printf("\n\nWelcome to our calculator program!\n");
}
```

Course Standard: Use a course header (`:hfx`) for each user-defined function. The `main` function does not require a header. The assignment header will act as the header for `main`.

This copyrighted text licensed to on via

A Function with no Parameters that Returns a Value

- Input functions are a good example of the category of functions which accept no data into the function but is able to generate and return a single value.

```
#include<stdio.h>

//FUNCTION DECLARATIONS

void welcome(void);
int getInput(void);

int main(void)
{
  //LOCAL VARIABLE DECLARATIONS
  int operand1;  //FIRST OPERAND ENTERED BY USER
  int operand2;  //SECOND OPERAND ENTERED BY USER

  //EXECUTABLE STATEMENTS
  welcome();

  operand1 = getInput();  //THE ASSIGNMENT OPERATOR AWAITS THE
  operand2 = getInput();  //VALUE RETURNED FROM getInput

  return(0);
}
```

Why does the order in which the functions are called matter? Why not add the new function calls to be first among the executables?

Why consider at this point in development the addition of a print statement to display the values of the integers after the new function calls?

What would happen if you printed these two integers before the new function calls?

This copyrighted text licensed to on via

Adding the definition of the input function to our work in progress:

```
/*****+-**--***-*---*-*---------**-*****--*******************************
*
*  Function Information
*
*  Name of Function:  getInput
*
*  Function Return Type:  int
*
*  Parameters (list data type, name, and comment one per line):  NONE
*
*  Function Description:  Accept and return user integer input.
*
******+-**--***-*---*-*---------**-*****--*******************************/
int getInput(void)
{
  //LOCAL DECLARATIONS
  int op;  //USER'S OPERAND INPUT - ONLY VARIABLE AVAILABLE TO THIS FUNCTION

  printf("Enter integer operand -> ");
  scanf("%d", &op);

  return(op);
}

//COURSE FUNCTION HEADER REMOVED FROM THIS FUNCTION TO SAVE SPACE

void welcome(void)
{
  printf("\n\nWelcome to our calculator program!\n");
}
```

Observations:

- The value input by the user is returned to the calling function and completes the assignment expression back in `main`.
- Notice the function declaration lists a return type of `int` and the lack of parameters indicated by empty parentheses.

This copyrighted text licensed to on via

A Function with Arguments that Returns a Value

```
#include<stdio.h>

//FUNCTION DECLARATIONS

void welcome(void);
int getInput(void);
int calcRemainder(int, int);
int calcQuotient(int, int);

int main(void)
{
  //LOCAL VARIABLE DECLARATIONS

  int operand1;  //FIRST OPERAND ENTERED BY USER
  int operand2;  //SECOND OPERAND ENTERED BY USER
  int remain;    //HOLDS THE RESULT OF THE REMAINDER
  int quotient;   //HOLDS THE RESULT OF INT DIVISION

  //EXECUTABLE STATEMENTS

  welcome();

  operand1 = getInput();  //THE ASSIGNMENT OPERATOR AWAITS THE
  operand2 = getInput();  //VALUE RETURNED FROM operandInput

  remain = calcRemainder(operand1, operand2);
  quotient = calcQuotient(operand1, operand2);

  return(0);
}
```

This copyrighted text licensed to on via

This copyrighted text licensed to on via

```
/*****+-**--***-*---*-*---------**-*****--*********************************
*
*  Function Information
*
*  Name of Function:  calcRemainder
*
*  Function Return Type:  init
*
*  Parameters (list data type, name, and comment one per line):
*    1.  int op1 - FIRST OPERAND (NUMERATOR) FOR REMAINDER CALCULATION
*    2.  int op2 - SECOND OPERAND (DENOMINATOR) FOR REMAINDER CALCULATION
*
*  Function Description:  Returns the result of the modulus operator for
*                         two integer values.
*
******+-**--***-*---*-*---------**-*****--*********************************/
int calcRemainder(int op1, int op2)
{
  int rem;  //TEMPORARY STORAGE OF REMAINDER

  rem = op1 % op2;

  return(rem);
}

//COURSE FUNCTION HEADER REMOVED FROM OLD FUNCTIONS TO SAVE SPACE

int getInput(void)
{
  //LOCAL DECLARATIONS
  int op;  //USER'S OPERAND INPUT - ONLY VARIABLE AVAILABLE TO THIS FUNCTION

  printf("Enter integer operand -> ");
  scanf("%d", &op);

  return(op);
}
```

```
void welcome(void)
{
  printf("\n\nWelcome to our calculator program!\n");
}

/*****+-**--***-*---*-*********---**-*****--*******************************
*
*  Function Information
*
*  Name of Function:  calcQuotient
*
*  Function Return Type:  int
*
*  Parameters (list data type, name, and comment one per line):
*    1.  int op1 - FIRST OPERAND FOR DIVISION CALCULATION (NUMERATOR)
*    2.  int op2 - SECOND OPERAND FOR DIVISION CALCULATION (DENOMINATOR)
*
*  Function Description:  Returns the result of the division operator for
*                         two integer values.
*
******+-**--***-*---*-*********---**-*****--*******************************/
int calcQuotient(int op1, int op2)
{
  return(op1 / op2);
}
```

Note: Any local declarations (`rem` variable in the `calcRemainder` definition) are still commented to the right and will NOT be included with the parameters in the course function header.

Observations:

- In `calcRemainder` a new local variable is created to store the value of the modulus operator then the value of the variable is returned.
- In the `calcQuotient` function the result of the expression is returned and is not stored locally in the function.
- The order in which function definitions are listed does not matter. **However,** `main` **must come first!!**

This copyrighted text licensed to on via

A Function with Arguments that Returns no Value (`void`)

- **Reminder:** A function is **`void`** if it returns no value. The term "void function" says nothing about parameters, but only the lack of a value returned.

```
//FUNCTION DECLARATIONS

void welcome(void);
int getInput(void);
int calcRemainder(int, int);
int calcQuotient(int x, int y);
void displayResults(int, int, int, int);
```

Note on identifiers in function declarations:

```
int main(void)
{
  //LOCAL VARIABLE DECLARATIONS
  int operand1;  //FIRST OPERAND ENTERED BY USER
  int operand2;  //SECOND OPERAND ENTERED BY USER
  int remain;    //REMAINDER GENERATED BY MODULUS
  int quotient;  //QUOTIENT GENERATED BY DIVISION

  //EXECUTABLE STATEMENTS
  welcome();

  operand1 = getInput();
  operand2 = getInput();

  remain = calcRemainder(operand1, operand2);
  quotient = calcQuotient(operand1, operand2);

  displayResults(operand1, operand2, remain, quotient);

  return(0);
}
```

Observations about `main` function:

This copyrighted text licensed to on via

This copyrighted text licensed to on via

```
/*****+-**--***-*---*-*--------**-*****--*******************************
*
*  Function Information
*
*  Name of Function:  displayResults
*
*  Function Return Type:  void
*
*  Parameters (list data type, name, and comment one per line):
*    1. int op1 - first operand for quotient and modulus calculation
*    2. int op2 - second operand for quotient and modulus calculation
*    3. int rem - result of modulus calculation
*    4. int quo - result of division calculation
*
*  Function Description:  Displays final result of calculator program.
*
******+-**--***-*---*-*--------**-*****--*******************************/
void displayResults(int op1, int op2, int rem, int quo)
{
  printf("%d / %d = %d\n", op1, op2, quo);
  printf("%d %% %d = %d\n", op1, op2, rem);
}
```

```
int calcRemainder(int op1, int op2)
{
  int rem; //TEMPORARY STORAGE OF REMAINDER

  rem = op1 % op2;

  return(rem);
}

int calcQuotient(int op1, int op2)
{
  return(op1 / op2);
}
```

Observations:

- Only the new `displayResults` function definition is included with its header to save space in the notes. Each and every one of your user-defined functions must include a completed course function header!
- Notice the declaration of this new function. Between the parentheses the type and number of the arguments for the function are listed.
- The number, type, and order of the parameters are important! But any parameter names (identifiers) found in a function declaration are optional and non-binding.

This copyrighted text licensed to on via

```
int getInput(void)
{
  //LOCAL DECLARATIONS
  int op;

  //EXECUTABLE STATEMENTS
  printf("Enter integer operand -> ");
  scanf("%d", &op);

  return(op);
}

void welcome(void)
{
  printf("\n\nWelcome to our calculator program!\n\n");
}
```

Reminder: The `void` in the first line of the function definition is optional.

CRITICAL PROGRAMMING STANDARD - From the course standards.

With our ability to implement user-defined functions **only the following will be permitted** in the `main` function:

1. Declaration of variables to be passed to functions.
2. Calls to user-defined functions by `main`.
3. A limited amount of control structures (see chapters 5 and 6) to retain the previous two tasks within the `main` function.

Additionally, each user-defined function may represent a single task in your larger program. The failure to make a good use of user-defined functions as described here and in the course standards will result in a loss of ALL points on those assignments that require user-defined functions.

Parameter Passing

When we send data to a function what is happening in memory of the computer?

The two possibilities are:

1. The values being passed to the function are stored in new variables representing new and unique memory locations. The result is essentially **making copies** of the original values.
2. The variables in the called function are linked to the variables in the calling function such that they **refer to the same memory** location. Changes made in the called function alter the variables in the calling function.

```
void exchangeValues(int, int);

int main(void)
{
  int x = 3;
  int y = 5;

  printf("Before the function: x = %d y = %d\n", x, y);

  exchangeValues(x, y);

  printf("After the function: x = %d y = %d\n", x, y);

  return(0);
}

void exchangeValues(int a, int b)
{
  int temp;

  printf("Before the exchange: a = %d b = %d\n", a, b);

  temp = a;
  a = b;
  b = temp;

  printf("After the exchange: a = %d b = %d\n", a, b);
}
```

Output Generated:

- If possibility #1 is true, then changing `a` will not change `x` as evidenced by the final print statement in `main`.
- If possibility #2, then upon changing the variable `a` in the called function we will observe the variable `x` being changed when control is returned to `main`.

This copyrighted text licensed to on via

Based on the results of the previous program we are left to conclude that it is only a copy of the value of the variable that is passed from calling function to the called function. This particular paradigm of passing data to a function is commonly called **passing by value.** The concept of data protection between functions finds its basis in the pass by value approach. The calling function stores the value received in a new memory location and is free to make changes to its copy of the value without worry about how such changes may result in a logical error back in the calling function.

Is the outcome of the previous program changed if the corresponding variables had the same name AND data type?

- Using the same data type in calling and called function is **not** that which determines whether the data is passed by value.

What roles do the identifiers of the variables in the calling and called function play in the pass by value paradigm?

- Using the same identifier in calling and called function is **not** that which determines whether a value is passed by value.

Pass by Value (downward communication) versus Pass by Address (upward communication)

A few questions that you should be pondering at this point are:

1. Is it ever possible to "return" more than one value from a function?
2. Are there are alternatives to pass by value?

The answer to both questions is YES and involves the same idea. What we need to be able to do is not pass a copy of the value of a parameter, but rather to make a connection between a variable in the called function to a variable back in the calling function. To do this, we must **pass the memory address of the variable** to a function rather than passing a copy of that variable's value. By passing the memory address of a variable in the calling function to the called function we can now make changes to the contents of that memory address and making any such changes is permanent and remains available to the calling function after the called function terminates.

To use pass by address we need to know a little bit about **pointers**. We will cover pointers in detail in chapters 9 and 10, but using them to accomplish pass by address will benefit us now.

Pointer Operations

- To pass the address of a single variable we use the address (`&`) operator before the name of that variable.
- A **pointer** is a variable that **stores the memory address** of another variable as its value. We need such a variable to receive the address being sent from the calling function to the called function.

This copyrighted text licensed to on via

Modification of previous example to use Pass-by-address Technique

```
#include<stdio.h>

void exchangeValues(int*, int*);

int main(void)
{
  int x = 3;
  int y = 5;

  printf("Before the function: x = %d y = %d\n", x, y);

  exchangeValues(&x, &y);

  printf("After the function: x = %d y = %d\n", x, y);

  return(0);
}

void exchangeValues(int *a, int *b)
{
  int temp;

  printf("Before the exchange: a = %d b = %d\n", *a, *b);

  temp = *a;
  *a = *b;
  *b = temp;

  printf("After the exchange: a = %d b = %d\n", *a, *b);
}
```

- **Note:** A function that makes use of **pass by address** does NOT have to be `void`, a value still can be returned through the use of the keyword `return`. Neither is such a function required to accept all parameters by address.

This copyrighted text licensed to on via

Describe the three assignment statement expressions between the two print statements in the `exchangeValues` function:

- `temp = *a;` Assign the value in the location to which `a` refers to the `temp` variable.
- `*a = *b;` Retrieve the value in the location which `b` refers and assign it to the location to which `a` refers.
- `*b = temp;` Assign the value of `temp` to the location to which `b` refers.

When to use pass by address?

When more than one value needs to be revised and that revision be retained in the calling function once the called function terminates.

AND

The function continues to meet the definition of being functionally cohesive such that it completes only a single fundamental task.

This copyrighted text licensed to

Scope

The **scope** of an object (think variable or function) is the region of the program for which it is defined and in which you can refer to it by its declared identifier. Can two or more objects have the same identifier? The answer is yes, if the **scope** of each object does not overlap! In many cases the reuse of an identifier with overlapping scope is a compiler error. In other situations it is a poor design choice.

Two levels of scope are used in this course, **global** and **local**. All **functions** will be declared **globally** and all **variables** will continue to always be declared **locally**.

Structure Charts

A **structure chart** is the primary design tool for a program. You should complete your structure chart for a program before you attempt to develop any flowcharts or write any code. A structure chart will represent each user-defined function that is present in your program and the order in which they are called. You can next create a flowchart for each function after your structure chart is complete.

on

Structure Chart Rules and Symbols

via

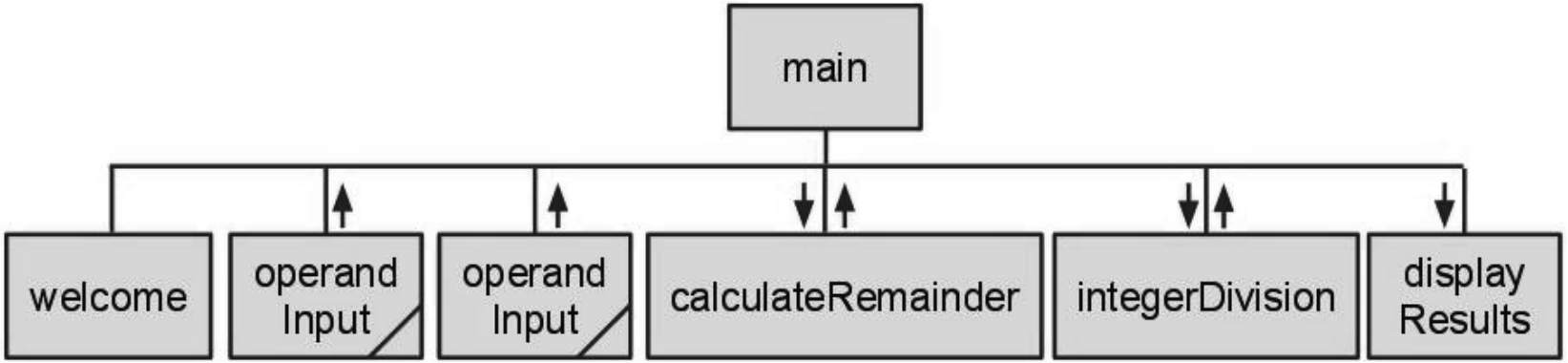
	The two symbols to the left are used to indicate function calls. The first symbol will represent a function called a single time in a program. The name of the function will go inside of the box.
	The second symbol contains a "cross-hatch" in the lower-right corner. All functions listed with such a cross-hatch indicate that it is called more than once in the program. To limit the number of sub-functions in a structure chart you only need to list the sub-functions called (if any) by this particular function one time (the first time it is used).

How to Read a Structure Chart

Structure charts are read top-down and then left-right. The book is inconsistent when it comes to what should be in the top-most box, we will recommend that this box be a representation of the `main` function. A structure chart shows only the order in which user-defined functions are called and that data exchanged, it will contain no code!

The following is a summary of the basic **structure chart rules:** (1) Each rectangle in a structure chart represents a user-defined function. Standard C functions (`stdio.h`, `math.h`) are not included. (2) The name in the rectangle should communicate the purpose of the function. It is the name that will be used in the program. (3) The structure chart contains only the function flow, no other code is indicated. (4) Common functions, those functions called more than once in a program, are indicated by a slash in the lower right corner of the rectangle. If such a function contains other function calls then the complete structure need be shown only once.	
	You may find it helpful in certain circumstances to list the values going into and out from a user-defined function. The symbols on the left can be used to designate such values going in and out of a function. Variables **sent to a function should be listed to the left** of the line and those **coming out of a function on the right.**

Structure Chart from calculator problem demonstrated in the notes:



How does the design of the above structure chart represent the expectation regarding the role of the `main` function in a well-designed program?

- In general, the structure charts are more horizontal (wide) than they are vertical (deep). This indicates that most, if not all, of the user-defined functions should be called from within the `main` function.
- As mentioned previously, every user-defined function should be limited to completing a single task of the larger problem which your program is designed to solve.

This copyrighted text licensed to on via

This copyrighted text licensed to on via

Flawed Design #1:	**Flawed Design #2:**
Concerns with the above design: • Insufficient number of functions. AND • One function to contain all tasks in the program. • The `main` function does nothing other than start the program. OR • One function to contain a trivial task in the program. • The `main` function completes the remaining tasks.	**Concerns with the above design:** • The `main` function does nothing other than start the program. • Lack of understanding regarding either the `return` statement or passing parameter techniques.

Introduction to Problem Solving Method

1. **Specify what is desired of the program.**
   - In an introductory programming course the specifications are given to you by the instructor assigning the problem.
   - In the real world you would meet with fellow engineers, scientists, supervisors, and/or clients to determine how and what the software must successfully accomplish.
   - **Recommendation:** Get experience working in this area before you graduate from college. Try programs such as EPICS, ENTR, and professional experiences.
2. **Analyze the problem.**
   - Factor the tasks as specified by top-down design making use of tools such as **structure charts.**
   - This step requires the development of the mathematical formulas necessary to solve the problem. Some formulas are very common such as determining the slope of a line, the distance between two points, the length of the hypotenuse of a right triangle, but other formulas are advanced and specific to your engineering or scientific discipline.
   - Once you have determined the arithmetic expressions necessary to solve the problem, be mindful to consider potential errors that may arise. Examples include dividing by zero or dealing with negative values where such is impossible (number of people, certain measures of distance and time). **Through planning you begin to work out potential errors in your logic.**
3. **Design the software solution.**
   - The flow chart or pseudo-code representation of the algorithm is developed at this point in the process. Both flowcharting and pseudo-coding are techniques you can use to design your algorithm to solve the problem at hand.
   - **Application:** When working with teams be sure to represent your logic visually so others can comprehend your thinking as you present an idea for a solution.
   - As our ability to solve problems becomes more advanced be sure to design the solution to each task individually. When you have all of the tasks designed the work of implementing those tasks to create a final solution is much easier.
4. **Translate the design into code.**
   - Code only small portions of the solution, usually a single function at a time.
   - Test each and every change you make!
   - Never write an entire program prior to compiling and testing for the first time!
   - **Benefit:** Investing time in the previous steps in the process can really make this step: (1) shorter in terms of time and (2) a lot less frustrating.
5. **Test and debug the implementation.**
   - When testing, be sure to develop test cases and expected output that go beyond those provided on the assignment.
   - What tests have you run to ensure that your program handles all reasonable input?
   - General cases, boundary cases, extreme/stress cases.
6. **Refinement of the solution.**
   - Congratulations, you have a solution! Now, review your solution and make sure it is efficient, well-designed, meets all requirements, produces no warnings or syntax errors, and satisfies all course documentation and programming standards.
     - **Your first solution is rarely the best solution. Make every effort to refine your algorithm so that it makes the best possible use of the limited resources of the computer.**

This copyrighted text licensed to on via

Structured Programming – What is it?

- There are three fundamental ideas in structured programming; selection, repetition, and sequence.
- **Sequence** was the topic of chapter 4 (user-defined functions). The use of functions permit the individual instructions of a program to be executed in an order other than first statement to the last statement and some statements may be executed more than once.
- **Selection** is the topic of chapter 5. Selection constructs will allow instructions to be executed based on a given criteria and other instructions to be excluded. This is the first point in the semester where not all code in a program may be executed in each test case.
- **Repetition** is the topic of chapter 6. Often a set of instructions needs to be repeated while a particular condition is met, or until a condition is met. Repetition constructs allow the use of logic to specify conditions and to identify those instructions to repeat.

Chapter 5 – Selection

While functions permit us to make modular code and shift control (alter **sequence** of code executed) between functions to accomplish individual tasks, what we need next are the abilities to **select** and **repeat** code. It is **selection** that is the topic of chapter 5.

Up to this point in the course we have always observed all statements in a program being executed at least one time.

- While user-defined functions allow us to alter the sequence at this point in the course when we make a function call all of the code in that function is executed.
- The idea of selection involves selecting code to execute or exclude based on a condition.

Logical Data

A piece of data is called **logical** if it conveys the value of true or false. Logical data is generated by a **logical expression**. Logical expressions are utilized in both selection and repetition constructs. The C programming language did not traditionally have a data type specifically for logical data and instead would utilize a numeric type to represent true and false.

Note: We often use 1 to represent a true value, however, ANY non-zero value is considered true.

This copyrighted text licensed to on via

Logical Operators

There are three main logical operators. Operands to logical operators are considered for their logical status (true or false).

Operator	Character(s)	Description	Precedence
NOT	!	Unary operator that changes a true value to false and a false value to true.	
AND	&&	Binary operator that is true only when both of its operands are true and is false in all other cases.	
OR	||	Binary operator that is false only when both of its operands are false and is true in all other cases.	

This copyrighted text licensed to

Logical Operators Truth Table:

not		and			or		
x	!x	x	y	x && y	x	y	x || y
TRUE		TRUE	TRUE		TRUE	TRUE	
		TRUE	FALSE		TRUE	FALSE	
FALSE		FALSE	TRUE		FALSE	TRUE	
		FALSE	FALSE		FALSE	FALSE	

on via

Evaluating Logical Expressions

- That is, evaluating expressions that contain logical operators.
- When evaluating mathematical expressions it would be the rare exception that you would not consider every operator.
- How can we evaluate an expression prior to evaluating every operator of the expression? With logical expressions there are only two possible results and we will see that when the result of the expression can be determined the evaluation will stop, or short-circuit, prior to evaluating every operator when doing so is unnecessary.

The short-circuit method:

Example (similar to) Program 5-1 from text:
`int a = 1;`
`int b = -1;`
`int c = 0;`
 	
`printf("%2d && %2d = %2d\n", a, b, a && b);`
`printf("%2d && %2d = %2d\n", a, c, a && c);`
 	
`printf("%2d || %2d = %2d\n", a, b, a || b);`
`printf("%2d || %2d = %2d\n", a, c, a || c);`
 	
`printf("%2d && %2d = %2d\n", a, !c, a && !c);`
`printf("%2d || %2d = %2d\n", !a, c, !a || c);`
 	
`printf("--a && !c = %2d\n", --a && !c);`
`printf("b++ || ++c = %2d\n", b++ || ++c);`
 	
`printf("a = %d b = %d c = %d\n", a, b, c);`
	Output:
	
	1.
	2.
	
	3.
	4.
	
	5.
	6.
	
	7.
	8.
	
	9.

Why do we observe the values we do for the final print statement in the code segment above?

This copyrighted text licensed to on via

This copyrighted text licensed to

Relational and Equality Operators

There are four relational operators and two equality operators found in the C programming language. Each of these operators are binary operators in which two operands are evaluated. **The result of a relational expression is a logical (0, 1) value.**

Relational and Equality Operators (figure 5-4, page 236):

Type	Operator	Meaning	Precedence
Relational	<	Less than	10
	<=	Less than or equal to	
	>	Greater than	
	>=	Greater than or equal to	
Equality	==	Equal to (equality)	9
	!=	Not equal to (inequality)	

Compound Statements

It is possible to combine a series of relational, equality, and logical operators to form what is called a **compound statement.** The result of such a statement (or expression) will be a (true or false) logical value.

The meaning of some compound logical expressions is clear when they contain only a single variable such as `x > 0 && x < 10`

Often a compound statement is made easier to read by including parentheses. Can you determine which of the expressions below is different than the other two?

Example:

```
int x = 3;
int y = 2;
int z = 3;

printf("Result #1 = %d\n", x <= 3 || y >= 2 && z != 3);
printf("Result #2 = %d\n", x <= 3 || (y >= 2 && z != 3));
printf("Result #3 = %d\n", (x <= 3 || y >= 2) && z != 3);
```

Note: Our compiler will issue a warning suggesting that you make use of (and) when using `&&` and `||` in the same expression.

on via

Extending the previous example:

```
int x = 3;
int y = 2;
int z = 3;

printf("Result = %d\n", x++ <= 3 || y++ >= 2 && z++ != 3);

printf("x: %d y: %d z: %d\n", x, y, z);
```

What is the output of the above program?	**What is operator binding? How is this related to precedence?** • The level of precedence for an operator can determine in an expression those operands that belong bound to it. • Because the logical AND operator has precedence over the logical OR operator the expression `y++ >= 2` is the left-hand side of the AND operator and not the right-hand side of the OR operator.

Example:

```
int x = 7;
int y = 9;
int z = 5;

int result;

result = !(++y - 9) || -5 + z++ && x++;

printf("x: %d y: %d z: %d\n", x, y, z);
```

How is it evaluated?

This copyrighted text licensed to on via

Complements

Each of the relational, equality, and logical operators is a complement of another relational, equality, or logical operator. Whenever one logical expression is true its complement would be false. Whenever one logical expression is false its complement would be true.

- For example, `x < 3` represents all numbers less than 3, the complement `x >= 3` represents all other values.

Complementing the Relational and Equality Operators:

Complementing the Logical Operators:

This copyrighted text licensed to on via

Original Expression:	Complement:	
`x > 0 && x < 10`		
`x > 0 && x < 10 || y != 0`		
`!(x != 1 && x != 2)`	Complement #1:	Complement #2:
`!x && !y`	Complement #1:	Complement #2:
`x == 0 || y == 0`	Complement #1:	Complement #2:

Selection Flowchart Example #1:

Ask the user for their exam score and based on their score let them know what grade they earned (A, B, C, D, F based on 90, 80, 70, 60) and give them an appropriate message of congratulations (or not).

Step #1 – What is specified of the program?

- Input – A single score.
- Output – A message related to the grade earned and the grade earned.

Step #2 – Analyze the problem

- Structure chart representing the user-defined functions of this problem.

This copyrighted text licensed to on via

This copyrighted text licensed to on via

- **Note:** In general it is best to eliminate any code that is common to more than one path in a selection construct. Such code should be placed before or after the selection construct. Every path in a selection construct should contain statements that are unique to that particular condition.

Two-Way Selection

- Logical expressions evaluate to either a true or false value.
- If the resulting logical value is true we have one path to travel, but if the result is false we will execute alternative (and unique) statements found down a second path.
- Two selection constructs are available for two-way selection in C programming: **`if/else`** and the **conditional expression**.

This copyrighted text licensed to on via

if / else

- The if/else construct, or a very similar variation, exists in about every programming language. The idea is that a logical expression is evaluated with one set of statements to be executed when the logical expression is true and an alternative set of statements to be executed when it is false.

```
if(logical expression)
{
  //TRUE ACTIONS TO EXECUTE
}
else
{
  //FALSE ACTIONS TO EXECUTE
}
```

- If the first condition (logical expression) listed is evaluated to be true the statements found in the body of the `if` will be executed, otherwise execute the statements found in the body of the `else`.

Diagram of Two-way Logic:

Example: A function receives a non-negative integer, add one and return it when even or subtract one and return it when odd.

```
int changeNumber(int n)
{
  if(n % 2 == 0)
  {
    n++;
  }
  else
  {
    n--;
  }

  return(n);
}
```

Example #2: Revise the previous function to only change the value (subtract one) if odd.

```
int changeNumber(int n)
{
  if(n % 2 == 1)
  {
    n--;
  }
  else
  {
    //NEVER LEAVE A BODY EMPTY
  }

  return(n);
}
```

```
int changeNumber(int n)
{
  if(n % 2 == 0)
  {
    //NEVER LEAVE A BODY EMPTY
  }
  else
  {
    n--;
  }

  return(n);
}
```

Best solution option for revised problem (`else` **is optional**):

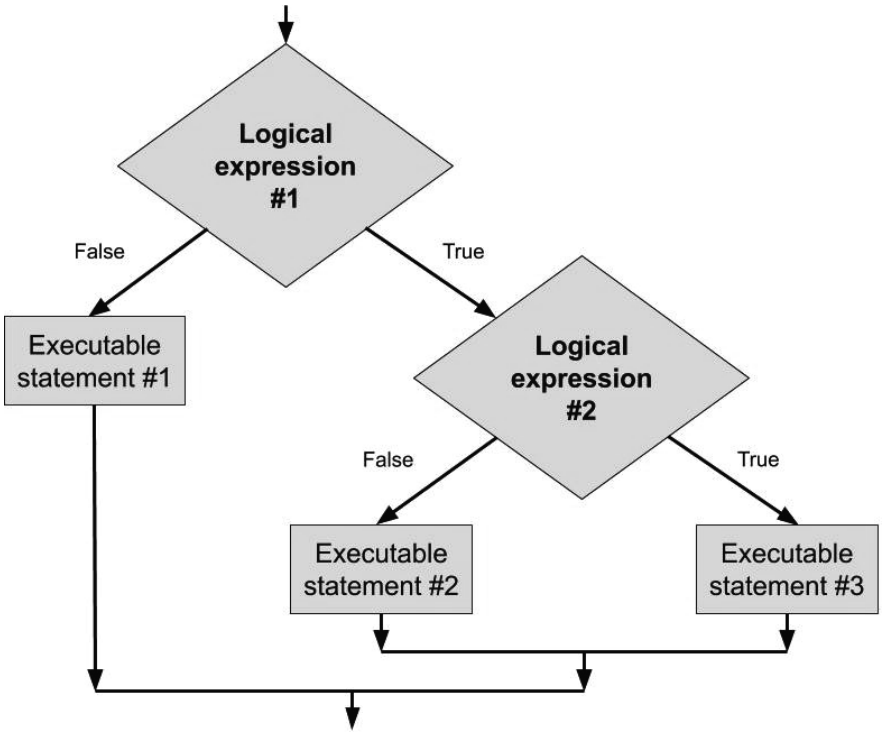
```
int changeNumber(int n)
{
  if(n % 2 == 1)
  {
    n--;
  }

  return(n);
}
```

This copyrighted text licensed to on via

Nested Selection

The `if/else` can contain any executable statement, including another `if/else` construct. When this occurs we call it **nested selection**. Often, nested selection involves the comparison of two or more different variables.

Logical Flow of Nested Selection:



Code example of nested selection:

```
int calcStatus(float gpa, int credit_hrs)
{
  int status;

  if(credit_hrs > 120)
  {
    if(gpa >= 3.50)
    {
      status = HONORS;
    }
    else
    {
      status = CANDIDATE;
    }
  }
  else
  {
    status = NONCANDIDATE;
  }

  return(status);
}
```

Dangling Else Logical Error and Nested `if/else`

- **Debugging Concern:** Please note that the text and many other sources will not always use `{ }` with `if/else` statements that contain only a single statement in their body. However, this can lead to a difficult logical problem known as the "dangling else".
- Because this problem can be so difficult to debug, **we will require ALL `if/else` statements to use `{ }` even if the body contains only a single executable statement.**

This copyrighted text licensed to on via

Example of dangling else logical error:	**Output Generated:**

`int x = 3;`
`int y = 2;`
`int z = 0;`
 	
`if(x == 3)`
`  if(y != 2)`
`  z++;`
`else`
`  z--;`
 	
`printf("z = %d\n", z);`
	`warning: suggest explicit braces to avoid ambiguous 'else'`

Logic interpretation of code above:

This copyrighted text licensed to on via

Conditional Expressions

The conditional expression has three operands and two operators. Each operand is itself an expression.

- The first operator is a **question mark** that separates the first two expressions.
- The second operator is a **colon** and separates the last two expressions.

To evaluate a conditional expression, we first must evaluate the leftmost expression. When this expression is true, then the value of the second expression (the first expression after the `?`) is evaluated. However, if the logical expression (before the `?`) is false, then the third expression is evaluated.

Below, when the sum of the two variables is greater than zero the variable `a` is incremented. When the sum is less than or equal to zero then the variable `b` is incremented.

```
a + b > 0 ? a++ : b++;
```

More Conditional Expression Examples:

```
int a = 1;
int b = -1;
int c;

a + b > 0 ? a++ : b++;

c = a > b ? 5 : 3;

printf("c = %d\n", c);

a == b ? printf("a == b\n") : printf("a != b\n");
```

Output Generated:

Must there be an expression for both the second and third expressions?

- Yes.
- `error: expected expression before ';' token`

Is it possible for each option, true and false to have more than one expression?

- No.

This copyrighted text licensed to on via

Multi-way Selection

It is a common problem to require selection from more than two alternative sections of unique code. This is the idea of multi-way selection.

- In C, there are two different constructs available to implement multi-way selection. The first is by using `if/else if/else`, the second is a `switch` construct. The `switch` construct has a few more restrictions than the `if/else if/else` construct but the `switch` does have circumstances in which its use might be favored over the alternative option.

Multi-way Selection `if/else if/else`

- **Example:** Write a program that will accept a score as input and reply to the user based on the score what letter grade they can expect to receive.

```
#define MINA 90
#define MINB 80
#define MINC 70
#define MIND 60

int getScore(void);
char determineGrade(int);
void displayResults(char, int);
```

```
int main(void)
{
  int score;  //SCORE OF STUDENT
  char grade; //CALCULATED GRADE BASED ON SCORE

  score = getScore();
  grade = determineGrade(score);

  displayResults(grade, score);

  return(0);
}
```

```
void displayResults(char grade, int score)
{
  printf("Based on your score of %d, you will receive the grade of %c in this course.\n", score, grade);
}

int getScore(void)
{
  int score;  //SCORE TO BE ENTERED BY THE USER

  printf("Enter the score for the course: ");
  scanf("%d", &score);

  return(score);
}
```

This copyrighted text licensed to on via

```
char determineGrade(int score)
{
  char grade;

  if(score >= MINA)
  {
    grade = 'A';
  }
  else if(score >= MINB)
  {
    grade = 'B';
  }
  else if(score >= MINC)
  {
    grade = 'C';
  }
  else if(score >= MIND)
  {
    grade = 'D';
  }
  else
  {
    grade = 'F';
  }

  return(grade);
}
```

Figure 5-24 on page 261 is a good flowchart illustration of this selection construct.

Note: In this example as soon as one of the conditions is true the statements in the body for the particular condition are executed and then the `if/else if/else` structure is exited.

This copyrighted text licensed to on via

Are either of the constructs below logically equivalent to the one on the previous page?

```
if(score >= MINA)
{
  grade = 'A';
}
else if(score >= MINB && score < MINA)
{
  grade = 'B';
}
else if(score >= MINC && score < MINB)
{
  grade = 'C';
}
else if(score >= MIND && score < MINC)
{
  grade = 'D';
}
else
{
  grade = 'F';
}
```

```
if(score >= MINA)
{
  grade = 'A';
}
if(score >= MINB && score < MINA)
{
  grade = 'B';
}
if(score >= MINC && score < MINB)
{
  grade = 'C';
}
if(score >= MIND && score < MINC)
{
  grade = 'D';
}
else
{
  grade = 'F';
}
```

What is the reason the AND along with the condition that follows is unnecessary in the three logical expressions above (left)?

This copyrighted text licensed to on via

Multi-way Selection `switch`

A `switch` construct is used to make a selection among possible integer value alternatives. The `switch` may be favored when the possible integer values are well known and few in number.

The **control expression** that follows the keyword `switch` is evaluated and then compared with a series of cases for equality. Each `case` represents an expected value of the control expression.

- Associated with each `case` is one or more executable statements.
- There should exist one or more possible cases. You should consider an `if/else if` when the number of cases is large or represent a range of values.
- A `default` case is an option to cover all values not represented by the other individual cases.

> **Note:** Each case is compared with the value of the control expression for equality.

```
void displayResults(char grade, int score)
{
  printf("Given score of %d, you will earn a grade of %c in this course.\n", score, grade);

  switch(grade)
  {
    case 'A':  printf("Great job!!\n");
               break;
    case 'B':  printf("Well done!\n");
               break;
    case 'C':  printf("Mission Accomplished.\n");
               break;
    case 'D':  printf("Perhaps you need to consider repeating?\n");
               break;
    default:   printf("See you next semester?\n");
  }
}
```

This copyrighted text licensed to on via

Course Standard: Please note how the indentation is handled in a `switch` construct.

Why are the values being compared with the control expression in single quotes?

The data type of the control expression is character (`char`) and to compare the results of such an expression with a character literal constant requires that the value be placed inside of single quotes.

If we were comparing integer values then NO single quotes would be used.

Why the need for the `break` statements?

```
switch(grade)
{
  case 'A':  printf("Great job!!\n");
  case 'B':  printf("Well done!\n");
  case 'C':  printf("Mission Accomplished.\n");
  case 'D':  printf("Perhaps you need to consider repeating?\n");
  default:   printf("See you next semester?\n");
}
```

Without the `break` statement as soon as one true `case` is found all following code will be executed!

```
Enter the score for the course: 94
Based on your score of 94, you will receive the grade of A in this
course.
Great job!!
Well done!
Mission Accomplished.
Perhaps you need to consider repeating?
See you next semester?
```

Course Standard: This is the ONLY application within the course that the use of a control forcing statement such as `break` is permitted.

Why is it challenging to test two floating-point values for equality?

- Representing floating-point data is not exact and is likely to contain round-off errors.
- The smallest difference at some seemingly insignificant decimal place will result in the equality operator generating a false value.
- Try the following and modify to identify other values that do or do not work as expected:

```
float x = 0.1;

x == 0.1 ? printf("True!\n") : printf("False!\n");
```

This copyrighted text licensed to on via

Switch Construct Rules

1. The control expression must be integral.
2. Each case label is the keyword case followed by a constant expression.
3. No two case labels can represent the same value.
4. Two case statements can be associated with the same executable statements.
5. The default label is not required.
6. There is a maximum of one default label.

Example of Rule #4, the first statement is printed if value of `grade` `'A'` or `'B'`

```
switch(grade)
{
  case 'A':
  case 'B':  printf("Great job!!\n");
             break;
  case 'C':  printf("Mission Accomplished.\n");
             break;
  case 'D':  printf("Perhaps you need to consider repeating?\n");
  default:   printf("See you next semester?\n");
}
```

Revision of `switch` above using the ASCII values (how to compare integer data within a `switch` construct):

```
switch(grade)
{
  case 65:
  case 66:  printf("Great job!!\n");
            break;
  case 67:  printf("Mission Accomplished.\n");
            break;
  case 68:  printf("Perhaps you need to consider repeating?\n");
            break;
  default:  printf("See you next semester?\n");
}
```

This copyrighted text licensed to on via

Misapplication of `switch`:	**Improvement by reducing number of cases:**

`switch(score)`
`{`
`  case 100:`
`  case  99:`
`  case  98:`
`  case  97:`
`  case  96:`
`  case  95:`
`  case  94:`
`  case  93:`
`  case  92:`
`  case  91:`
`  case  90: grade = 'A';`
`  break;`
`  case  89:`
`  case  88:`
	`switch(score / 10)`
	`{`
	`  case 10:`
	`  case  9:  grade = 'A';`
	`  break;`
	`  case  8:  grade = 'B';`
	`  break;`
	`  case  7:  grade = 'C';`
	`  break;`
	`  case  6:  grade = 'D';`
	`  break;`
	`  default:  grade = 'F';`
	`}`

Reminder:

- There is a natural temptation to only and always make use of the `if/else` with for all selection needs.
- Consider whether the use of a `switch` or a conditional expression might be a better choice in some situations. The result may be shorter and more readable logic.

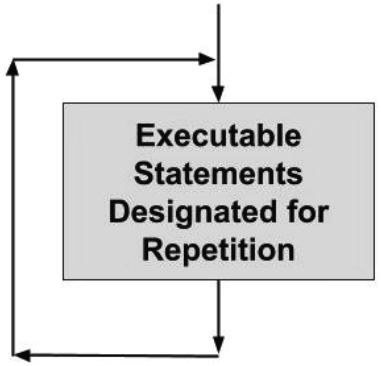
This copyrighted text licensed to on via

Chapter 6 – Repetition

The real power of computers is the ability to rapidly repeat an operation or series of operations. The act of such repetition is commonly referred to as **looping** but this term does not incorporate all techniques of repetition.

Looping Basics and Terminology

- If this diagram represents the logic of a loop then you might be wondering when, how, or if this process stops?



First, a few terms:

- loop - One of two possible techniques of repetition, utilizes a construct to repeat designated statements under a predetermined set of conditions.
- loop iteration - One execution of the instructions inside the body of the loop construct.
- loop control expression - The logical expression to determine whether the loop should initiate another iteration.
- loop initialization - Preparation work to be completed before the first evaluation of the loop control expression.
- loop update - An executable statement found inside the body of the loop that when executed a sufficient number of times will lead to a change in the evaluation of the loop control expression.
- loop control variable - The variable that is initialized, part of the loop control expression, and the recipient of the update action.

This copyrighted text licensed to on via

Repetition Flowchart Example #1:

Write an algorithm to calculate the average, maximum, and minimum of an unknown number of exam scores. The scores on the test can range from 0 – 100 and will all be integer values.

Repetition

- To complete this problem we need the ability to repeat a number of instructions.
- Repetition permits us to identify a segment of code to be repeated while a given logical expression is true.

Step #1 – What is specified of the program?

- Input – Integers between 0 and 100, inclusive of the end points, until a value is entered to signify the end of input.
  - We **do not** need an array (subject of chapter 8) to store the entire data set in the memory of the computer. The memory necessary to solve this problem **will not** be dependent on the size of the data set (number of students taking the exam).
- Output – Average (floating-point), maximum (integer), and minimum (integer) values.

Step #2 – Analyze the problem

- **What formulas are needed?** To calculate the average we need to know the sum of all scores and the number of scores. We will have to compare values as they are entered to determine their status as the maximum or minimum value.
- We also need to determine how the user will indicate that the input has ended.

How to approach the terminating input condition?

1. Ask the user to count the exams.
2. Ask the user to input after each value whether they have more scores to enter.
3. Ask the user to input a value outside of the range of exam scores when done.

Structure Chart for Solution:

This copyrighted text licensed to on via

Step #3 – The Software Solution

This copyrighted text licensed to on via

Pretest and Post-test Looping Constructs

This copyrighted text licensed to on via

In a **pretest** loop, the loop control expression must be true prior to iterating the loop even one time. Prior to the start of each iteration, including the first, the loop control expression is evaluated and while that expression remains true the repetition continues. If the loop control expression is false then no further iterations will be conducted.

Key Point: The minimum number of times the body of a pretest loop will be iterated is **zero**. If the expression is never true, then the loop instructions (body of the loop) will **never be executed**.

In a **post-test** loop the instructions of the body of the loop will be executed prior to the evaluation of the loop control expression. The loop control expression will be evaluated at the end of each iteration to determine whether or not to continue with another iteration.

- **What is the big difference?** The minimum number of times the body of a post-test loop will be iterated is one. Regardless if the expression is false the loop will iterate once and then test the condition to determine whether iteration continues.

Selecting the Wrong Construct

What signs might indicate that you selected the wrong type of loop, pretest or post-test, for a solution? (1) Logic repeated both inside and outside of the body of the loop. (2) A "trick" is used to force the loop to iterate at least one time. These options might look familiar if you have worked with a language in which both pretest and post-test options are not available (MATLAB, Python).

Example: The factorial of a number is the product of all integers from 1 to the number. Design a function that given a non-negative integer will calculate its factorial.

Factorial flow chart:

This copyrighted text licensed to on via

Identify the loop control variable:	**Identify the type of loop:**	**Identify how the loop control variable is updated:**	**Is the algorithm logically correct when the user enters zero?**

Event-Controlled and Counter-Controlled Processes

The factorial problem from the previous page is an example of a **counter-controlled process**. The counter, which is also the loop control variable, controls the number of times the body of the loop is to be repeated.

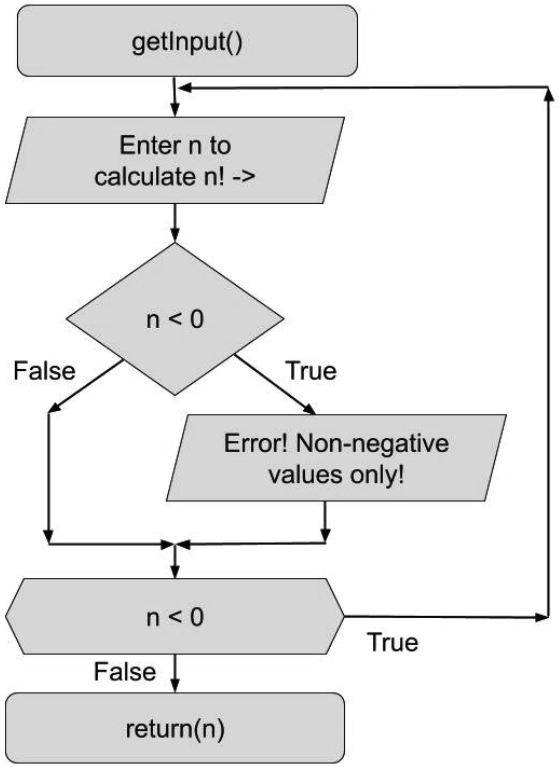
In an **event-controlled process**, an event takes place that changes the loop control expression from true to false and bringing the repetition to a halt. With such a process one can determine whether another iteration of the loop will take place but determining the number of remaining iterations is not possible. The exam statistics problem is an example of an event-controlled process. If you know the most recent input then you know if there will be another iteration, but nothing more.

Key Point: If you can count or calculate the total number of iterations that will take place during the execution of a loop, then it is a **counter-controlled** process, otherwise the process is **event-controlled**.

Input Validation

- The goal of **adding code to our input functions** is to ensure that before the function is allowed to terminate that we have a meaningful value to process in the remainder of our program.
  - The task of input validation is **commonly found within the input function**. Going forward from this point input functions will be more than print-scan-return statements.
- You must always provide the user with an **unlimited number** of opportunities to enter meaningful data.
- Due to its complexity you will **not validate for particular data types** only for acceptable values within a specified data type.
  - You will assume that the user will always enter the expected data type as input.
- One important part of input validation is to display an **appropriate message** to the user making them aware of their mistake and that input which is desired.
- Because input validation does make input functions more complex it is **ill advised** to use **pass by address** in an effort to "return" multiple values from a single input function. The syntax associated with pass by address and related pointer operations (chapters 9 and 10) make it challenging to apply to input validation.

Flowchart for input validation function:



This copyrighted text licensed to on via

Loops in C

The C programming language provides three constructs for the looping technique of repetition. These are; `while`, `for`, and `do-while`. We will begin with the similar `while` and `do-while` constructs.

The `while` Loop

The `while` construct is a **pretest** loop. As with all loops the `while` makes use of a loop control expression to determine whether the loop should begin or continue to iterate. Being a **pretest** loop the loop control expression is evaluated prior to the first iteration of the loop, then while the expression is evaluated to be true then the loop will complete another iteration.

- The iterations will continue WHILE the loop control expression is true. When the control expression evaluates to false the iterative process stops.

Basic Syntax of the While Loop

```
while(loop control expression)
{

  //EXECUTABLE STATEMENT DESIGNATED
  //FOR REPETITION - BODY OF THE LOOP

}
```

Note: There is no semicolon after the loop control expression of the `while` statement, because the loop is followed a body which contains the actions to repeat.

The `do-while` Loop

The `do-while` construct is a **post-test** loop. Similar to the `while` a loop control expression is used to determine if repetition continues, but as is the case of a post-test construct the expression is evaluated after the initial execution of the instructions inside of the body.

- In contrast to the `while` loop, the keyword `while` in a `do-while` is terminated with a semicolon.

Course Standard: Indent all statements that compose the body of a loop **two additional spaces**.	**Course Standard:** With all loops (and `if` statements) the braces that mark the start and end of the loop body MUST be used.

Basic Syntax of the Do While Loop

```
do
{

  //EXECUTABLE STATEMENT DESIGNATED
  //FOR REPETITION - BODY OF THE LOOP

}while(loop control expression);
```


This copyrighted text licensed to on via

What happens if the update of the loop control variable occurs in the middle of body of the loop, will this change cause the loop to terminate immediately at this point?

- No. The loop will only terminate when the loop control expression is evaluated and found to be false.
- There is no "mid-test" loop in the C programming language, only pretest and post-test.

This copyrighted text licensed to on via

Factorial Example

- The factorial problem will permit a demonstration of both the `do-while` with input validation and the `while` to calculate the factorial.

```
int calcFactorial(int);
int getInput(void);
void displayResults(int, int);

int main(void)
{
  int num;
  int fact;

  num = getInput();
  fact = calcFactorial(num);
  displayResults(num, fact);

  return(0);
}
```

```
int calcFactorial(int num)
{
  int nFact = 1;

  while(num > 0)
  {
    nFact = nFact * num;
    num -= 1;
  }

  return(nFact);
}
```

Is there reason to be concerned that the value of zero may be multiplied?

- No, the loop control expression is evaluated and false prior to the multiplication of zero.

The value entered by the user is stored locally in the variable `num`, how does changing `num` on each iteration influence what is going on back in the calling function?

- No, the variable has been passed by value.

```
int getInput(void)
{
  int num;

  do
  {
    printf("Enter n to calculate n! -> ");
    scanf("%d", &num);

    if(num < 0)
    {
      printf("\nError! Non-negative values only!!\n\n");
    }
  }while(num < 0);

  return(num);
}

void displayResults(int num, int fact)
{
  printf("\n%d! = %d\n", num, fact);
}
```

Why don't we use an "if loop" for input validation?

- An `if` construct would only permit the user a second opportunity to enter acceptable data. We want the user to have an unlimited number of opportunities to enter valid data.

What is wrong with the term "if loop"?

- An `if` is a selection construct and not a repetition construct.

Nested Loops

What is a nested loop?

- Nested loops, or any form of nested repetition, is when one repetitive process is found inside of another.

How does that definition help solve problems that make use of nested loops?

- A better definition would be to consider that a loop solves a repetitive process. A nested loop would then be a repetitive process found inside of another repetitive process.

What is the recommended approach to solving nested looping problems?

- First, solve and test the nested process placing it in its own user-defined function.
- Next, solve the process that will call the nested process. We must ensure that the nested process is correct before we attempt to make use of (call) it repeatedly.

This copyrighted text licensed to on via

Calculating a Range of Factorials: Instead of calculating a single factorial the user will specify a range of values and desires the output of the factorial for all integers in the range.

For example, if the user enters 2 and 5 then they want to see the result of 2!, 3!, 4!, and 5!.

- What are the two repetitive processes here? One is the calculation of a single factorial and the other is counting from the starting value to the ending value.
- Which process is the nested process? The calculation of a single factorial. We have already confirmed that we are able to accept an integer and calculate the factorial of that number.
- Next we need to add to the existing program such that we can calculate each `n!` for all integers `n` in the range from `a` to `b`.
- New tasks or revisions needed:
  - Get the `start` and `end` values and validate both such that the `end` is not less than the `start`. **Why is there a parameter for `getEnd` but none for `getStart`?**
  - Calculate and print the factorial for each integer between `start` and `end` (inclusive).

```
int calcFactorial(int);                 //UNCHANGED FROM PREVIOUS EXAMPLE
int getStart(void);
int getEnd(int);
void displayResults(int, int);          //UNCHANGED FROM PREVIOUS EXAMPLE

int main(void)
{
  int fact;
  int start;
  int end;

  start = getStart();
  end = getEnd(start);

  while(start <= end)
  {
    fact = calcFactorial(start);
    displayResults(start, fact);
    start++;
  }

  return(0);
}
```

- **Course Standard:** A limited amount of control structures (selection, repetition) is permissible in the `main` function to ensure that it is the `main` function which makes most of the function calls for the program.

This copyrighted text licensed to on via

User-defined function for input of starting value:

```
int getStart(void)
{
  int num;

  do
  {
    printf("Enter starting value -> ");
    scanf("%d", &num);

    if(num < 0)
    {
      printf("\nError! Non-negative values only!!\n\n");
    }
  }while(num < 0);

  return(num);
}
```

User-defined function for input of ending value:

```
int getEnd(int start)
{
  int num;

  do
  {
    printf("Enter ending value -> ");
    scanf("%d", &num);

    if(num < start)
    {
      printf("\nError! Ending value must be at least %d\n\n", start);
    }
  }while(num < start);

  return(num);
}
```

This copyrighted text licensed to on via

Where are the nested loops? It is common to first consider the examples below when one thinks of **nested** constructs.

Nested `while` loop example:

```
while(  )
{
  while(  )
  {
  }
}
```

Nested `if` construct example:

```
if(  )
{
  if(  )
  {
  }
}
```

In the nested loop example provided on the previous page the loops are found in two different user-defined functions. One is in the `main` function that counts from the starting point to the ending point and the second is found in the function that calculates the factorial for each integer in the desired range.

Every loop typically represents a task, or single process, in a program. If you can identify each of these individual tasks and factor them into individual user-defined functions you may find the complex problem easier to solve and test.

The Infinite Loop

The infinite loop is a logical error in which the condition of the loop control expression will never be false. As a logical error the form in which infinite loops may appear while the program is running can vary. One sure sign of an infinite loop is the production of infinite output. However, the error can appear in forms in which little or no output is generated.

- You can terminate a program at any time by entering `CTRL-C` from the keyboard.

Example: Allow the user to enter integer values until a 1 or 2 is input.

```
int getInteger(void)
{
  int num;

  do
  {
    printf("Enter 1 or 2 -> ");
    scanf("%d", &num);
  }while(num != 1 || num != 2);

  return(num);
}
```

Output:

```
Enter 1 or 2 -> 3
Enter 1 or 2 -> 2
Enter 1 or 2 -> 1
Enter 1 or 2 ->
```

- The control expression may seem logically correct, “accept input while the value entered is not 1 or 2”, but logically it results in an infinite loop. There is no single value that cannot be **both** not 1 and not 2 at the same time.

This copyrighted text licensed to on via

Repetition Review - Three Common Loop Components

1. **Initialization** of the loop control variable.
2. **Evaluation** of the loop control expression.
3. **Update** to the loop control variable.

The `for` loop

The third and final looping construct available in the C programming language is the pretest `for` loop. In addition to being a pretest looping construct the `for` loop was designed to be used only with counter-controlled processes. The `for` loop brings together the initialization, loop control expression, and update to a single line of code. With this benefit it will be a course standard to always make use of all three expressions of a `for` loop.

A `while` loop representation using all three expressions:	**Flowchart representation of a `for` loop:**

This copyrighted text licensed to on via

Expression 1 (Initialization)

- Expression 1 is evaluated only when the `for` loop is encountered for the first time. It is in this expression that the loop control variable will be initialized.

Expression 2 (Evaluation)

- After expression 1 is evaluated we immediately evaluate expression 2 which is the loop control expression.
- If expression 2 is initially evaluated to be false then the loop will not iterate! The `for` loop is a pretest construct.
- Upon completion of every iteration the loop control expression must be re-evaluated to determine whether another iteration is needed.

Expression 3 (Update)

- This final expression is evaluated at the end of every iteration and before the next evaluation of expression 2.
- Essentially, any expression changing the loop control variable is acceptable for the third expression.

Syntax

- Semicolons are necessary between expression 1 and 2 and between expression 2 and 3.
- As with any looping construct the `for` loop has a body and the body will begin and end with `{` and `}`. The body of instructions can be one or more instruction.

When to use a `for` loop?

Otherwise… consider a `while` for your pretest iteration needs.

This copyrighted text licensed to on via

This copyrighted text licensed to on via

Without altering the initialization values, convert the `while` loop into a `for` loop:

```
int lcv = 1;
int n = 4;
int nFact = 1;

while(lcv <= n)
{
  nFact *= lcv;
  lcv++;
}
```

Converted loop:

Without altering the initialization values, convert the `while` loop into a `for` loop:

```
int lcv;
int n = 4;
int nFact = 1;

lcv = n;

while(lcv > 0)
{
  nFact *= lcv;
  lcv--;
}
```

Converted loop:

What potential concerns exist with the application of the `for` loop below?

```
int x;

for(x = -1; x < 0; )
{
  printf("Enter a non-negative integer: ");
  scanf("%d", &x);
}
```

Rearranging Digits in a Number Example:

- Given an integer, rearrange its digits to create the largest number possible.

Example Executions:

```
Enter an integer: 1230789
The sorted digit is 9873210

Enter an integer: 53635
The sorted digit is 65533
```

Step #1 – What is specified of the program?

- We expect an integer to be entered and to be output.
- Leading 0's do not make sense as integer input and will not be considered.
- All input will be non-negative and must be validated.

Step #2 – Analyze the problem

- How do we extract a single digit from a number?
- How do we do this and then determine if it is the largest digit in the number?
- What do we do once we have the largest digit in the number?
- What about finding the second largest digit in the number?

Step #3 – Designing the Software Solution

What are the tasks of this program?

1. Get the input from the user.
2. Count the number of non-leading zero digits so we can append them to the value output at the end.
3. Find the largest digit in the input. "Add" it to a new number. "Delete" it from the value input.
4. Return to step #3 until ALL non-zero digits, largest to smallest, have been moved from the value input to the new integer value being constructed.
5. Append the zero digits on to the end of the new value.
6. Output.

This copyrighted text licensed to on via

Structure Chart

This copyrighted text licensed to on via

`main` Function

`getInput` **Function**

`ctZero` **Function**

This copyrighted text licensed to on via

`findMax` Function

This copyrighted text licensed to on via

`appendMax` Function

`removeMax` Function

Solution:

```
int getInput(void);
int ctZero(int);
int findMax(int, int*);
int appendMax(int, int);
int removeMax(int, int, int);
void output(int);

int main(void)
{
  int n;
  int sorted = 0;
  int max;
  int maxLoc;
  int zeroCount;

  n = getInput();
  zeroCount = ctZero(n);

  while(n > 0)
  {
    max = findMax(n, &maxLoc);
    sorted = appendMax(sorted, max);
    n = removeMax(n, max, maxLoc);
  }

  output(sorted * (int)pow(10, zeroCount));

  return(0);
}

int appendMax(int sorted, int max)
{
  return(sorted * 10 + max);
}

int removeMax(int n, int max, int maxLoc)
{
  return(n - max * (int)pow(10, maxLoc));
}
```

This copyrighted text licensed to on via

```
int getInput(void)
{
  int n;

  do
  {
    printf("Enter an integer to sort: ");
    scanf("%d", &n);

    if(n < 0)
    {
      printf("\nError! Non-negative values only!\n\n");
    }

  }while(n < 0);

  return(n);
}

int ctZero(int n)
{
  int ct = 0;

  while(n > 0)
  {
    if(n % 10 == 0)
    {
      ct++;
    }

    n = n / 10;
  }

  return(ct);
}


void output(int sorted)
{
  printf("Sorted value: %d\n", sorted);
}
```

```
int findMax(int n, int *maxLoc)
{
  int max = 0;
  int loc = 0;

  while(n > 0)
  {
    if(n % 10 > max)
    {
      max = n % 10;
      *maxLoc = loc;
    }

    n = n / 10;
    loc++;
  }

  return(max);
}
```

This copyrighted text licensed to on via

Advanced Topic – Recursion

Recursion is the alternative technique of repetition. The definition of recursion applies to a function which calls itself. Solutions to repetition problems that use loops are referred to as **iterative** solutions. As with loops the function calls cannot continue infinitely. The condition when another call is to be made is the **recursive case**. When no more calls are to be made then the **base case** has been reached.

Remember, control of a program always returns from the called function to the calling function. In addition to the control of the program a value may be returned as well.

This copyrighted text licensed to on via

The **factorial** is one of the most common first examples given to demonstrate the use of recursion.

Consider how a factorial for a number `n` is calculated: `n * (n - 1) * (n - 2) * ... * 2 * 1`

You could also say that `n!` is equivalent to `n * (n - 1)!` for `n > 1`, but what about when `n = 1` ? It is at this point where we find our base case or a definition that when `n = 1` then `n! = 1`.

Let's take `4!` as an example:

```
4! = 4 * 3!
3! = 3 * 2!
2! = 2 * 1!
1! = 1
```

Solving the factorial with the `while` loop:

```
int factorial(int n)
{
  int nFact = 1;

  while(n > 0)
  {
    nFact *= n--;
  }

  return(nFact);
}
```

Solving the factorial using a recursive function:

```
int factorial(int n)
{
  int nFact = 1;

  if(n > 1)
  {
    nFact = n * factorial(n - 1);
  }

  return(nFact);
}
```

Trace the execution of the recursive solution when the value 4 is passed to the factorial function. List the value of the variables in each call. Note the progression of the control of the program as the recursive function calls are made.

```
int factorial(int n)
{
  int nFact = 1;

  if(n > 1)
  {
    nFact = n * factorial(n - 1);
  }

  return(nFact);
}
```

```
int factorial(int n)
{
  int nFact = 1;

  if(n > 1)
  {
    nFact = n * factorial(n - 1);
  }

  return(nFact);
}
```

```
int factorial(int n)
{
  int nFact = 1;

  if(n > 1)
  {
    nFact = n * factorial(n - 1);
  }

  return(nFact);
}
```

```
int factorial(int n)
{
  int nFact = 1;

  if(n > 1)
  {
    nFact = n * factorial(n - 1);
  }

  return(nFact);
}
```

This copyrighted text licensed to on via

Note: Calling a function is different than an iteration of a loop where the same code inside of the loop is being executed repeatedly. Each function call is independent of other function calls with local variables occupying unique memory locations.

Review of Repetition Constructs and Techniques

This copyrighted text licensed to on via

Limitations of Recursion

In the factorial example we saw that calculating 4! required four function calls, in each function call a unique set of variables is created and the status of each function needs to be "remembered" in order to continue after the called function terminates. A large number of function calls can require more memory than the computer is capable of providing.

When should you NOT use recursion?

- When the number of recursive function calls has the potential to be large (counter-controlled) or unpredictable (event-controlled).

Chapter 8 – Arrays

The data types introduced up to this point in the course only have the capacity to store a single value. Repetition in the previous chapter opened the possibility to accepting and processing a large data set. How does that work with our existing data types?

Problem: A recent exam was taken by ten students and we'd like to calculate maximum, minimum, and average. We know from a previous example that we can calculate maximum, minimum, and average without needing to store all of the scores in variables at the same time, but to calculate the median and mode we do need to keep all values and sort those values to assist in making these calculations.

One possibility may include declaring variables for each value to be entered.

```
int score0;
int score1;
int score2;
int score3;
int score4;
int score5;
int score6;
int score7;
int score8;
int score9;
```

The task of input and output for ten individual variables will require either ten print and scan statements, one print and scan statement with ten placeholders, or calling a function ten times or with ten parameters. **The amount of code should not depend on data set size!**

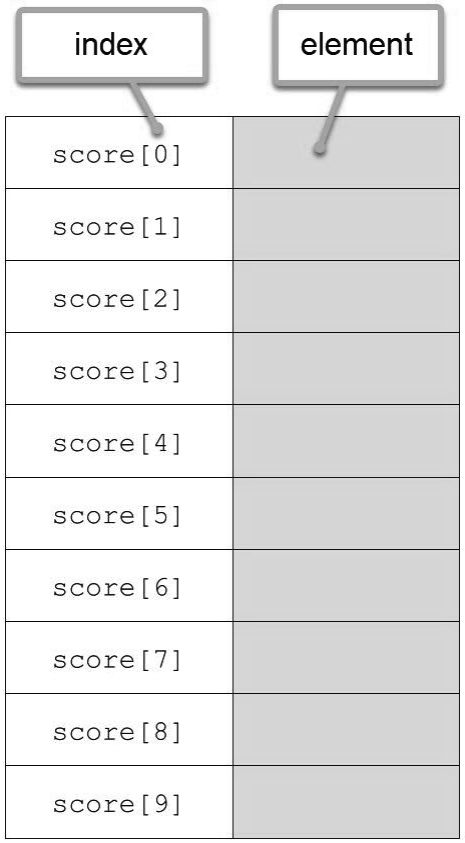
Perhaps it is possible to manage this small number of variables, but what if I had 100 students? Or even 1,000 students?

- What is needed is a more practical way to store and access a collection of related data items.

An **array** is a **consecutive** series of **variables** that share one name and data type. Each variable, or **element,** of an array has a number known as an **index value** that represents its position in the array. When we reference the name of the array along with a given index value we are referring to a specific individual value stored in that array.

This copyrighted text licensed to on via

Arrays of scores:



Notice that this ten-element array has a range of index values that begins at zero and continues to one less than the size of the array. In the C programming language **arrays are zero-based**.

At each index there is the capacity to store one value of the declared type.

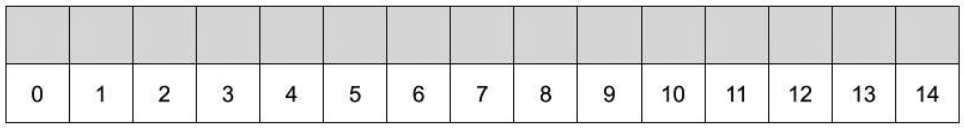
Declaring and Defining Arrays

As with other variables we must declare and define arrays before they can be used in a function. Declaration and definition inform the compiler of the **name** of the array, the **size** of the array, and the **data type** of each element. The C programming language provides for two different array types; fixed-length arrays (length is known before program is compiled) and variable-length arrays (size is determined when program is running).

Course Standard: All arrays will be **static** (of fixed-length) until we introduce the topic of **dynamic** memory allocation in chapters 9 and 10. The application of variable-length arrays as seen in the example on pages 478-479 **will violate course standards** as it permits the local declarations and executable statements in a function to overlap.

- **Note:** it will be a standard for the course to replace the number (size) between `[` and `]` with a **symbolic constant** for statically declared arrays.

```
int students[15];
```




```
double gpa[50];
```

This copyrighted text licensed to on via

Initialization of Arrays

- What is stored in the memory of a newly declared array?

An array can be initialized as part of its declaration statement:	**What if we attempted the following initialization?**
`int data[5] = {9, 5, 1, 4, 2};`	`int moreData[500] = {1, 2, 3};`
Note: If the number of values provided is less than the defined size of the array, the unassigned elements are filled with zeros.	

Accessing Elements in Arrays

- An **index** value is used to access individual **elements** in an array.
- The index value must be an integral value which may include an expression that evaluates to such.

Assignment of Values into an Array

Write a program that will assign the score of 9.5 to the first element in the array `labScores`. Next, print that element to the screen.

```
#define NUMLABS 10

int main(void)
{
  float labScores[NUMLABS];

  labScores[0] = 9.5;

  printf("First lab score: %.2f\n", labScores[0]);

  return(0);
}
```

This copyrighted text licensed to on via

Input of Values into an Array

```
#define NUMLABS 10

int main(void)
{
  float labScores[NUMLABS];

  printf("Enter first lab score -> ");
  scanf("%f", &labScores[0]);

  printf("First lab score: %.2f\n", labScores[0]);

  return(0);
}
```

Why the use of `%f` in the `printf` and `scanf` seen above?

- Individual elements of a `float` array are single `float` values.

If elements can only be accessed through the use of constants representing index values then we have the same problem we had before arrays, the amount of code we need depends on the quantity of data to process. It is more common to use a loop when handling many tasks related to multiple elements of an array.

This copyrighted text licensed to on via

Example using a loop to access the elements of an array:

```
#define NUMLABS 10

int main(void)
{
  int i;
  float labScores[NUMLABS];

  for(i = 0; i < NUMLABS; i++)
  {
    printf("Enter lab #%d score -> ", i + 1);
    scanf("%f", &labScores[i]);
  }

  //PROCESS DATA - FUNCTION CALLS RELEVANT TO PROBLEM BEING SOLVED WOULD GO HERE

  for(i = 0; i < NUMLABS; i++)
  {
    printf("Lab #%d score: %.2f\n", i + 1, labScores[i]);
  }

  return(0);
}
```

This copyrighted text licensed to on via

Index Range Checking

The valid range of index values for an array of size `NUMLABS` is zero to one less than `NUMLABS`. What happens if during the execution of the program an attempt is made to access index outside of this range?

For example, what if we tried to access `labScores[10]` when the only valid index values are 0 – 9 for an array declared to be of size 10? The scenario described here would imply that this potential problem is a run-time error. The compiler will not, and cannot, detect an index range checking problem.

Because this is a run-time error it can take several different forms. What are the possible consequences for attempting to access an index that is outside of the bounds of an array?

Possible Observation	Reason why this may occur?
No problem, program works as expected.	Going beyond the limits of the array and into memory that is NOT reserved for the purpose of the array MAY NOT be reserved for any other purpose.
Program executes, but results are unexpected.	Going beyond the limits of the array and into memory that is NOT reserved for the purpose of the array MAY be reserved for any other purpose and multiple edits to the same memory may leave an unexpected value being stored.
Program crashes.	Going beyond the limits of the array and attempting to access memory that DOES NOT exist or that the program does not have the ability to access.

Examples of exceeding the defined limits of an array

```
int x[5] = {8, 4, 5, 2, 3};
```

	8	4	5	2	3	
	0	1	2	3	4	

	8	4	5	2	3	
	0	1	2	3	4	

	8	4	5	2	3	
	0	1	2	3	4	

This copyrighted text licensed to on via

Arrays and Functions (Inter-Function Communication)

- Arrays are going to be of limited use to us if they are confined only to the function in which they are declared. There are two approaches for sharing data in an array with a user-defined function, we can either **send an element, or elements, or the entire array.**

Within each method we must understand the following:

1. How the **function declaration**, **function call**, and **function definition** change in terms of syntax.
2. Whether the values being passed are **by value** or **by address** and if a method other than the default can be used.

Passing Individual Elements

You can pass individual elements to a function like any other variable, as long as the array element type matches the function parameter type, it can be successfully passed to a function.

```
#define SIZE 5

int calcFactorial(int);

int main(void)
{
  int num = 8;
  int data[SIZE] = {2, 4, 6, 3, 5};

  printf("Result: %d\n", calcFactorial(num));

  printf("Result: %d\n", calcFactorial(data[2]));

  return(0);
}

int calcFactorial(int x)
{
  int xFact = 1;

  while(x > 1)
  {
    xFact *= x--;
  }

  return(xFact);
}
```

This copyrighted text licensed to on via

Example: Given an initialized integer array, print the square of each integer in the array.

```
#define ARRAYSIZE 5

void print_square(int);

int main(void)
{
  int lcv;
  int base[ARRAYSIZE] = {3, 7, 2, 4, 5};

  for(lcv = 0; lcv < ARRAYSIZE; lcv++)
  {
    print_square(base[lcv]);
  }

  printf("\n");

  return(0);
}

void print_square(int x)
{
  printf("%d squared = %d\n", x, x * x);
}
```

This copyrighted text licensed to on via

Observation from the previous program:

- When passing a single element of an array to a function you must specify the name of the array followed by the specific index of the value being passed in between the `[ ]` braces.

Review of Pass By Value

- Recall if we pass a single variable to a function that it is only a **copy** of the variable's value that is sent to the function such that changes to this **copy** does not change the variable back in the calling function.
- How is a single element of an array passed to a function? The first example that follows demonstrates the pass by value technique as demonstrated in chapter 4. The second example attempts to demonstrate whether a single element of an array is passed by value or by address.
  - Should `x[1]` be 24 in the final print statement then we would conclude that the element was **passed by value**.
  - Should `x[1]` be 3 in the final print statement then we would conclude that the element was **passed by address**.

This copyrighted text licensed to on via

```
void passTest(int);

int main(void)
{
  int x = 32;

  printf("Before x = %d\n", x);

  passTest(x);

  printf("After x = %d\n", x);

  return(0);
}

void passTest(int g)
{
  printf("Before g = %d\n", g);

  g = g / 8;

  printf("After g = %d\n", g);
}
```

Output of program above:

```
void passTest(int);

int main(void)
{
  int x[3] = {16, 24, 32};

  printf("Before x[1] = %d\n", x[1]);

  passTest(x[1]);

  printf("After x[1] = %d\n", x[1]);

  return(0);
}

void passTest(int g)
{
  printf("Before g = %d\n", g);

  g = g / 8;

  printf("After g = %d\n", g);
}
```

Output of program above:

From the output you can see that a single element of an array is treated the same as a single variable being passed to a function, that is, a **single element of an array is passed by value.**

- Is it possible to make use of pass by address with individual elements of an array?

Example: Passing the `x[1]` element by address.

```
#include<stdio.h>

void passTest(int*);

int main(void)
{
  int x[3] = {16, 24, 32};

  printf("Before x[1] = %d\n", x[1]);

  passTest(&x[1]);

  printf("After x[1] = %d\n", x[1]);

  return(0);
}

void passTest(int *g)
{
  printf("Before g = %d\n", *g);

  *g = *g / 8;

  printf("After g = %d\n", *g);
}
```

Example: Swap two elements of an array by passing them both by address to a function.

```
#define SIZE 10

void swapValues(int*, int*);

int main(void)
{
  int x[SIZE] = {3, 4, 5, 1, 2, 9, 8, 7, 6, 0};

  printf("x[4] = %d x[5] = %d\n", x[4], x[5]);

  swapValues(&x[4], &x[5]);

  printf("x[4] = %d x[5] = %d\n", x[4], x[5]);

  return(0);
}

void swapValues(int *ax, int *ay)
{
  int temp;

  temp = *ax;
  *ax = *ay;
  *ay = temp;
}
```

Passing the Whole Array

- What if we have an array of size 10 and we need to pass all 10 elements to a function?

Option #1, Pass Every Element

```
void fun(int, int, int, int, int, int, int, int, int, int);
```

Option #2, Pass Entire Array

```
void fun(int[]);
```


This copyrighted text licensed to on via

There is more here in terms of syntax and an end result that may be surprising when it comes to our conclusion regarding the passing technique used.

```
#include<stdio.h>

void passTest(int[]);

int main(void)
{
  int x[3] = {16, 24, 32};

  printf("Before x[1] = %d\n", x[1]);

  passTest(x);

  printf("After x[1] = %d\n", x[1]);

  return(0);
}

void passTest(int y[])
{
  printf("Before y[1] = %d\n", y[1]);

  y[1] = y[1] / 8;

  printf("After y[1] = %d\n", y[1]);
}
```

When passing the whole array to a function you only need to list the name of the array.

DO NOT include any of the following:

- The square braces as seen in the function declaration and definition.
- The size of the array as used in the array declaration.
- The data type of the array. When have we ever listed the type of a parameter in a function call?

Output of program on left:

Conclusion – Whole arrays are passed by address.

Notes:

- Consider this; if arrays were passed by value then this requires a complete duplication of memory necessary to store the copy of the array. This would be a very inefficient use of memory. Rather than passing a copy of the array the language passes the address of the array. **Is it even possible to pass a whole array by value?** No.
- No `&` operator in the user-defined function call as in chapter 4? **The name of an array already references a memory address!**
- **Which memory address is represented by the name of an array?** Arrays only track where they begin in memory and not where they end!
- **What does an index value represent?** The index value provided is added to the address represented by the name of the array to locate the desired element inside of the array. The index essentially serves as an offset from where the array begins in the memory of the computer.

This copyrighted text licensed to on via

Example: Calculate the average of the values in a 50-element integer array:

```
#include<stdio.h>

#define ARRAYSIZE 50

float average(int[]);
void getData(int[]);

int main(void)
{
  int base[ARRAYSIZE];
  float avg;

  getData(base);

  avg = average(base);

  printf("The average = %.2f\n", avg);

  return(0);
}
```

```
void getData(int b[])
{
  int lcv;

  printf("Enter integer data set now -> ");

  for(lcv = 0; lcv < ARRAYSIZE; lcv++)
  {
    scanf("%d", &b[lcv]);
  }
}

float average(int b[])
{
  int lcv;
  float avg = 0.0;

  for(lcv = 0; lcv < ARRAYSIZE; lcv++)
  {
    avg += b[lcv];
  }

  avg = avg / ARRAYSIZE;

  return(avg);
}
```

What would be the result of the print statement below if it were inserted into the `main` function ?

```
printf("base = %d\n", base);
```

Why are index range violations possible?

- The address represented by the name of the array is added to the specified index to calculate a new memory address. This address may be beyond that memory specifically allocated for the array.

This copyrighted text licensed to on via

Arrays and Measures of Efficiency

Write a function that will reverse the elements of an array (of `ARRAYSIZE`) so that the first element becomes the last and the last becomes the first, and so on...

Less Efficient Algorithm	More Efficient Algorithm
In what ways is the more efficient algorithm an improvement over the less efficient approach?	

This copyrighted text licensed to on via

Sorting Introduction

Often data requires sorting in order to be processed efficiently. This is evident when one considers data such as medical records, student academic records, or banking records. Without sorting data by some common field (last name or identification numbers) it would be difficult to locate and access a desired record when the number of patients, students, or bank customers becomes large.

Before we begin to look at the following simple sorting algorithms accept this disclaimer: The simplest sorting algorithms perform poorly. We introduce them to you as beginners because they are relatively easy to understand, write, test, and debug. Much more complex algorithms are often needed for good performance; such algorithms require data structures too complex for this course.

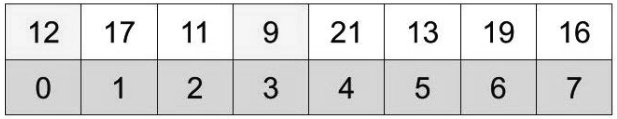
This copyrighted text licensed to on via

Exchanging Values

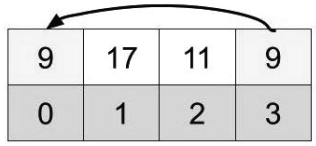
- Sorting may require values found to be out of order to be exchanged, or swapped, within the array. For example, exchange the two elements found at index 0 and index 3.

Incorrect Method of Exchanging Elements in an Array:

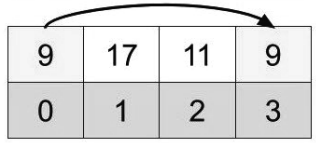
```
int x[8] = {12, 17, 11, 9, 21, 13, 19, 16};
```




```
x[0] = x[3];
```

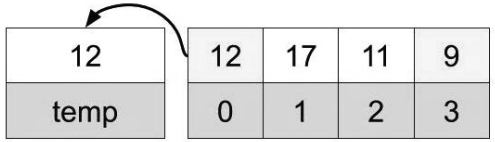



```
x[3] = x[0];
```

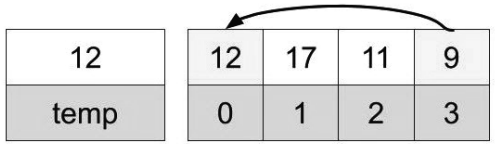


Correct Method of Exchanging Elements in an Array:

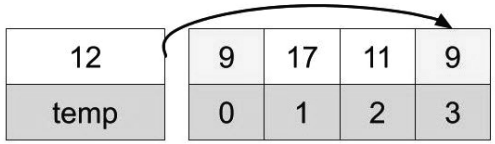
```
temp = x[0];
```

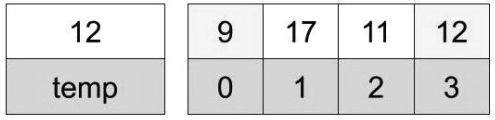



```
x[0] = x[3];
```




```
x[3] = temp;
```





There are a few common terms we need to introduce prior to explaining the sorting algorithms we will study this semester.

- **List** - The array will be separated into the unsorted list and sorted list. The goal is each algorithm is to move data from the unsorted list to the sorted list.
- **Pass** - The process of moving an element from the unsorted list to the sorted list. Several passes will be necessary to sort most data sets. [The final pass through the selection and bubble sorts will deviate from this general rule.]

Selection Sort

- **Description of algorithm:** Traverse the unsorted list to identify the smallest (or largest) value. Exchange this value with the value in the unsorted list that is adjacent to the sorted list. Repeat until the unsorted list is empty.

This copyrighted text licensed to on via

Bubble Sort

- **Description of algorithm:** Traverse the unsorted list and compare neighboring elements. Each time there is a pair of elements that are out of order, swap them.

  A pass constitutes comparing all neighboring elements in the unsorted list. In the selection sort on a single exchange was possible involving at most two elements, but in the bubble sort many swaps can occur on a single pass causing great change in the array.

This copyrighted text licensed to on via

Insertion Sort

- **Description of algorithm:** Compare the value in the unsorted list that is adjacent to the sorted list and determine where among the values in the sorted list it is to be inserted.

  The insertion sort is concerned with maintaining its sorted list in a sorted state. It is possible that a value once placed in the sorted list may move again on a future pass.

This copyrighted text licensed to on via

The Bubble Sort

Note: The code below is a variation of the Bubble Sort and is different from the code given in the book.

```
#define NUMDATA 10

void bubbleSort(int[]);
void getData(int[]);
void printData(int[]);

int main(void)
{
  int data[NUMDATA]; //ARRAY TO HOLD DATA TO SORT

  getData(data);
  printData(data);  //PRINT BEFORE STATE
  bubbleSort(data);
  printData(data);  //PRINT AFTER STATE

  return(0);
}

void printData(int x[])
{
  int lcv;  //LOOP CONTROL VARIABLE TO ACCESS ARRAY

  for(lcv = 0; lcv < NUMDATA; lcv++)
  {
    printf("x[%d] = %d\n", lcv, x[lcv]);
  }
}

void getData(int x[])
{
  int lcv;  //LOOP CONTROL VARIABLE TO GET DATA

  for(lcv = 0; lcv < NUMDATA; lcv++)
  {
    scanf("%d", &x[lcv]);
  }
}
```

This copyrighted text licensed to on via

```
void bubbleSort(int x[])
{
  int numPasses;  //LCV THAT CONTROLS # OF PASSES
  int lcv;        //LOOP CONTROL VARIABLE FOR SORTING
  int temp;       //HOLDS VALUE DURING SWAP

  for(numPasses = 1; numPasses < NUMDATA; numPasses++)
  {
    for(lcv = 0; lcv < NUMDATA - numPasses; lcv++)
    {
      if(x[lcv] > x[lcv + 1])
      {
        temp = x[lcv];
        x[lcv] = x[lcv + 1];
        x[lcv + 1] = temp;
      }//END OF IF
    }//END OF NEIGHBOR COMPARING LOOP
  }//END OF PASSES LOOP
}//END OF FUNCTION
```

Modify the Bubble Sort to work with an array that may not be filled to capacity:

```
void bubbleSort(int x[], int size)
{
  int numPasses;  //LCV THAT CONTROLS # OF PASSES
  int lcv;        //LOOP CONTROL VARIABLE FOR SORTING
  int temp;       //HOLDS VALUE DURING SWAP

  for(numPasses = 1; numPasses < size; numPasses++)
  {
    for(lcv = 0; lcv < size - numPasses; lcv++)
    {
      if(x[lcv] > x[lcv + 1])
      {
        temp = x[lcv];
        x[lcv] = x[lcv + 1];
        x[lcv + 1] = temp;
      }//END OF IF
    }//END OF NEIGHBOR COMPARING LOOP
  }//END OF PASSES LOOP
}//END OF FUNCTION
```

This copyrighted text licensed to on via

Summary Thoughts

How many passes are required to guarantee an array of `N` elements is sorted? • `N-1` passes would be necessary to complete the task of sorting.
Why is it possible for an array to reach a sorted state prior to the required number of passes to guarantee sorting? • The initial status of the data was such that fewer than `N-1` sort passes were needed to complete the task.
Are these sorting algorithms optimized to end when the data reaches a sorted state? • No. Could they be optimized? Possibly, we must be careful not to add logic for terminating early that requires more effort than just completing `N-1` passes.

How many elements are in the sorted listed of a 6 (six) element array for each of the algorithms below?

Pass Number	Selection / Bubble	Insertion
1		
2		
3		
4		
5		

This copyrighted text licensed to on via

Searching

Searching is the process used to find the location of a target in a data set. Perhaps the location, number of occurrences, or presence of the target in the data set provides us with more information about the target.

We will explore two different searching algorithms, the sequential search followed by the binary search. In addition to understanding the logic of each algorithm we will look at issues of efficiency and they compare to each other.

- In the **sequential search** we traverse the array one index at a time until we find the target or search the entire array. Assumptions about our data set include that every element is unique and the data is not sorted.

17	21	5	15	22	31	3	12	25	30	18	6	10	33	27
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

This copyrighted text licensed to on via

Searching Program #1 - Ask the user to enter a target value and return the location of that value in the array.

- If the value does not exist, please inform the user.

```
#define ARRAYSIZE 15
#define NOTFOUNDERROR -1

int searchArray(int[], int);
int getTarget(void);

int main(void)
{
  int array[ARRAYSIZE] = {17, 21, 5, 15, 22, 31, 3, 12, 25, 30, 18, 6, 10, 33, 27};
  int target;  //VALUE USER WISHES TO FIND
  int index;   //INDEX OF USER'S VALUE IN ABOVE ARRAY

  target = getTarget();
  index = searchArray(array, target);

  if(index == NOTFOUNDERROR)
  {
    printf("The value %d does not exist in the array!\n", target);
  }
  else
  {
    printf("The value %d was found at index %d of the array.\n", target, index);
  }

  return(0);
}
```

Why make use of the value -1 to determine that the target value was not found in the array? • The value is not a valid index of an array.
What happens when the target is not present in the data set? • To verify that this is the case every element in the array must be compared with the target.

This copyrighted text licensed to on via

```
int searchArray(int x[], int n)
{
  int lcv;                     //LOOP CONTROL VARIABLE USED TO ACCESS ARRAY ELEMENTS
  int index = NOTFOUNDERROR;   //VALUE TO RETURN WHICH INDICATES LOCATION OR NOT FOUND

  for(lcv = 0; lcv < ARRAYSIZE; lcv++)
  {
    if(x[lcv] == n)
    {
      index = lcv;
      lcv = ARRAYSIZE;
    }
  }

  return(index);
}
```

The user-defined searching function above will search every element of the array until one of the following events takes place:

- The target is found, the loop terminates, and the index value is returned.
- The end of the array is reached, the loop terminates, and the "not found" value is returned.

```
int getTarget(void)
{
  int target;

  do
  {
    printf("Please enter the target value: ");
    scanf("%d", &target);

    if(target < 0)
    {
      printf("Please enter a valid target value >= 0!\n");
    }
  }while(target < 0);

  return(target);
}
```

This copyrighted text licensed to on via

This type of search is known as the **sequential search** because each element of the array may potentially be referenced in order to find the desired element or to determine that element does not exist.

What is the goodness of the **sequential searching** algorithm?

- It works! It is simple.

What problems are associated with the use of the sequential searching algorithm?

What if we had a very large data set and the value we are looking for does not exist?
• It would take a long time to search every element and make this determination.
What if multiple instances of the target value are potentially present in the array?
• The entire array must now be searched to find every instance of a target in an unsorted array.

The Binary Search

- It is possible to reduce the time (both average and worst-case) it takes to complete a search by using an alternative searching algorithm known as the **binary search**. The binary search does require that the **data in the array be sorted**.

Is there a cost associated with sorting data?
• Yes, but if we search our data frequently it is worth paying the cost of sorting and maintaining our data in a sorted state.
Should the binary search always be used when searching?
• No, only when the data set is large and searched frequently.

New Integer Variables in the Binary Search

- `first` – This integer will represent the lowest index value at which the target value may potentially be found in the array.
- `last` – This integer will represent the highest index value at which the target value may potentially be found in the array.
- `mid` – The average of the `first` and `last` variables will determine where in the array the next comparison will be made with the target value. The `mid` is updated at the start of every iteration of the binary search process.
- How and when are the `first` and `last` variables updated? When the value at the `mid` is less than the target then the `first` is set to the value of the mid plus one. When the value at the `mid` is greater than the target then the `last` is set to the value of the mid minus one.

This copyrighted text licensed to on via

Binary Search Example

Assume we have the array given below, notice it is sorted and will remain sorted throughout the entire algorithm. A searching algorithm should never alter the data it searches! **Target = 25.**

3	5	6	10	12	15	17	18	21	22	25	27	30	31	33
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

This copyrighted text licensed to on via

What is the goodness of the **binary** search as applied to the example above?

- Only four comparisons were needed to find target (in the above example). A sequential search would have required 11.
- **Bigger Point:** With each comparison made between the target and the element at the midpoint we eliminate approximately half (hence the name binary) of the remaining index values as a possible location where the target could be found.

How does the binary search compare with the sequential search when it comes to finding a value that is not present in the array?

- The **sequential search** requires EVERY element of the array to be searched to be sure that the value does not exist.

Let's see in the example below how the binary search deals with a value that does not exist! **Target = 14.**

3	5	6	10	12	15	17	18	21	22	25	27	30	31	33
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

This copyrighted text licensed to on via

Searching Program #2 - Ask the user to enter a target value and return the location of that value in the array. If the value does not exist, please inform the user.

```
#define NOTFOUNDERROR -1
#define ARRAYSIZE 15

int binarySearch(int[], int);
int getTarget(void);                //SAME FUNCTION USED IN SEQUENTIAL SEARCH PROGRAM

int main(void)
{
  int array[ARRAYSIZE] = {3, 5, 6, 10, 12, 15, 17, 18, 21, 22, 25, 27, 30, 31, 33};
  int target; //VALUE WE ARE SEEKING
  int index;  //INDEX VALUE AT WHICH TARGET IS FOUND

  target = getTarget();
  index = binarySearch(array, target);

  if(index == NOTFOUNDERROR)
  {
    printf("Your value %d does not in the array!\n", target);
  }
  else
  {
    printf("Your value %d was found at index %d of the array.\n", target, index);
  }

  return(0);
}
```

This copyrighted text licensed to on via

```
int binarySearch(int x[], int n)
{
  int first = 0;              //SMALLEST INDEX VALUE WE ARE CONSIDERING
  int last = ARRAYSIZE - 1;   //LARGEST INDEX VALUE WE ARE CONSIDERING
  int mid;                    //MID POINT INDEX THAT WE ARE CONSIDERING
  int index = NOTFOUNDERROR; //VALUE TO RETURN WHICH INDICATES LOCATION OR NOT FOUND

  do
  {
    mid = (first + last) / 2;

    if(n > x[mid])             //TARGET GREATER THAN VALUE AT MID -> UPDATE FIRST
    {
      first = mid + 1;
    }
    else if(n < x[mid])        //TARGET LESS THAN VALUE AT MID -> UPDATE LAST
    {
      last = mid - 1;
    }
    else                       //TARGET HAS BEEN LOCATED IN THE ARRAY
    {
      index = mid;
      last = first - 1;        //WHAT DOES THIS DO?  WOULD first = last + 1 ALSO WORK?
    }
  }while(first <= last);

  return(index);
}
```

The user-defined searching function will search the array until the `first` is greater than the `last` which would occur for one of the following reasons:

- The target is found.
- There are no elements in the array remaining as candidates to be searched.

This copyrighted text licensed to on via

Multidimensional Arrays

The arrays we have seen so far are **single dimension** arrays. Each single dimension array is one row of indexed column values. Mastery of logic with single dimension arrays is necessary before considering multidimensional structures that may increase memory usage significantly.

With **multidimensional arrays** we can add a second dimension (multiple rows each with its own columns), a third dimension (planes, rows, and columns), and further as necessary. Your only limitation may be memory, but rarely does a problem call for an array with more than three dimensions.

Regardless of the number of dimensions in an array the name selected for the array will continue to represent the location where in memory its allocated space begins. We will see how this requires additional information to be communicated with functions that have a multidimensional array as a parameter.

Declaration and Defining Multidimensional Arrays

Multidimensional arrays, like single dimension arrays, must be declared and defined before being used. Declaration and definition tell the compiler the **name**, **data type,** and **size, or extent, of each dimension** of the array.

Data Type	Identifier	Dimension Extents
`int`	`table`	`[4][4];`

Two-dimensional array:

- Each dimension is zero-based.
- The intersection of every row and column represents the capacity to store a value of the defined type.
- Total capacity is the product of the extent of each dimension.

This copyrighted text licensed to on via

This copyrighted text licensed to on via

A three-dimensional array:

A four-dimensional array:

Problem, calculate the average of each row in a two-dimensional array:

```
#define ROWS 5
#define COLS 4

void printTable(int[][COLS], float[]);
void calcAvgs(int[][COLS], float[]);

int main(void)
{
  float avgs[ROWS];
  int table[ROWS][COLS] =
  {
    {1, 2, 3, 4},
    {2, 3, 4, 5},
    {3, 4, 5, 6},
    {4, 5, 6, 7},
    {5, 6, 7, 8}
  };

  calcAvgs(table, avgs);
  printTable(table, avgs);

  return(0);
}
```

Note: The extent (or size) of each dimension beyond the first **must** be explicitly listed in **BOTH** the function declaration and definition. The extent of the first dimension is optional (true for single dimension arrays).

How is a whole multidimensional array passed to a function?

- Name only.

By what method is this data passed to the function?

- Whole arrays, including multidimensional arrays, are passed by address.

Why is it required to specify the extent of the second dimension in the function declaration and definition as seen in this program?

- The name of an array only references where it begins in the memory of the computer. No information regarding dimensions is stored.

```
void calcAvgs(int data[][COLS], float avgs[])
{
  int currentRow;
  int currentCol;
  int sum;

  for(currentRow = 0; currentRow < ROWS; currentRow++)
  {
    sum = 0;

    for(currentCol = 0; currentCol < COLS; currentCol++)
    {
      sum += data[currentRow][currentCol];
    }

    avgs[currentRow] = (float)sum / COLS;
  }
}

void printTable(int data[][COLS], float avgs[])
{
  int currentRow;
  int currentCol;

  for(currentRow = 0; currentRow < ROWS; currentRow++)
  {
    printf("Row #%d: ", currentRow + 1);

    for(currentCol = 0; currentCol < COLS; currentCol++)
    {
      printf("%d ", data[currentRow][currentCol]);
    }

    printf("Average: %.2f\n", avgs[currentRow]);
  }
}
```

This copyrighted text licensed to on via

Chapters 9 and 10 – Pointers and Pointer Applications

- A **pointer** is a variable that stores a memory address, typically of another variable. and through the memory address the data stored there can be accessed and manipulated.

Working With Addresses, the & Operator

We have seen the address operator (`&`) required in `scanf` statements and being used to pass variables by address to functions, but what does this operator really do?

- The address operator (`&`) will return the address of the memory allocated to a variable. In the table below you can see two character variables `a` and `b`. Both of these variables are given a unique location in the memory of the computer to store a character value. The `printf` statement will display the memory address of `a` and `b` and does not print the contents of these variables.
- The `%p` placeholder is commonly used to print the address of a variable. The `p` conversion code represents a pointer value (memory address).
  - It is not necessary to place the `&` before the name of an array as the name of an array already represents a memory address.

```
char a = 'X';
char b = 'y';

printf("Address of a = %p\n", &a);
printf("Address of b = %p\n", &b);
```

```
Address of a = 0x7ffd7167170f
Address of b = 0x7ffd7167170e
```

Pointer Variables

- As with all variables we must declare a pointer before we can assign a memory address to it.
- The declaration of a pointer is quite similar to declaring a single variable, we need a data type and the name (identifier) of the variable with a `*` preceding the name to indicate that the variable is a pointer.

Sample local pointer declarations:

```
int *x;
float *y;
char *z;
```

Alternative and equivalent format:

```
int* x;
float* y;
char* z;
```

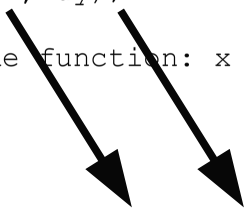

This copyrighted text licensed to on via

Where do the pointer variables, as previously declared, `x`, `y`, and `z` initially point?

- **Answer:** We do not know! We have seen how this can be bad with data variables already this semester; however, with pointers the unknown value may be an invalid memory address, then if the pointer is referenced before it is ever initialized the result will be a **segmentation fault**.

When passing by address the pointers are initialized as a result of the addresses that were sent from the calling function:

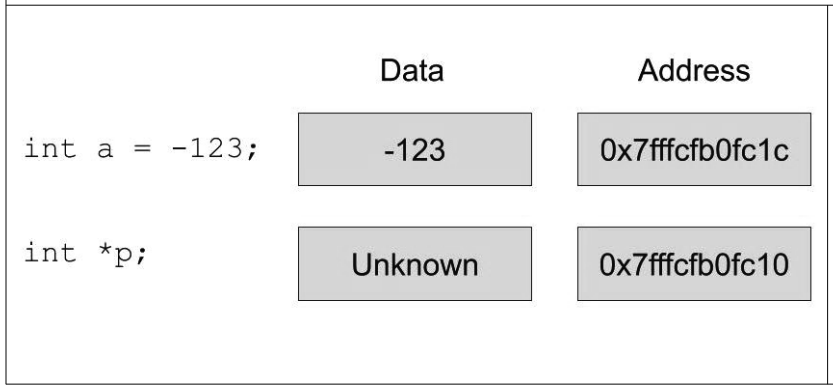
```
  exchangeValues(&x, &y);

  printf("After the function: x = %d y = %d\n", x, y);

  return(0);
}

void exchangeValues(int *a, int *b)
{
```



Initialization of Locally Declared Pointers

In order for a pointer to be initialized we must provide a meaningful place in memory for it to point (reference). Another variable is one valid place to which a pointer can point. We need to **assign** (=) a memory address to the pointer.

```
  int a = -123;
  int *p;

  printf("Address of a: %p\n", &a);
  printf("Address of p: %p\n", &p);

  p = &a;

  printf("Location to which p references: %p\n", p);
```



This copyrighted text licensed to on via

Now that we have established a meaningful place for the pointer variable to reference, what can be accomplished through the pointer?

The printing the value of `p` in the previous code produces the address to which `p` points and next we need a way to access the value stored in that memory location. The **indirection operator (*)** placed before the pointer variable will permit access to the value stored at the location to which the pointer references.

```
int a = -123;
int *p;

p = &a; //p POINTS TO a

printf("The value of a is: %d\n", a);
printf("The value to which p references is: %d\n", *p);
```

```
The value of a is: -123
The value to which p references is: -123
```

Example (Similar to figure 9-8, page 563 in text):

```
int a = 3;
int b;
int *x;
int *y;

x = &a;
y = x;

printf("a = %d *x = %d *y = %d\n", a, *x, *y);

*x = 4;

printf("a = %d *x = %d *y = %d\n", a, *x, *y);

x = &b;

printf("a = %d *x = %d *y = %d b = %d\n", a, *x, *y, b);

*x = a * *y;

printf("a = %d *x = %d *y = %d b = %d\n", a, *x, *y, b);
```

Output:

This copyrighted text licensed to on via

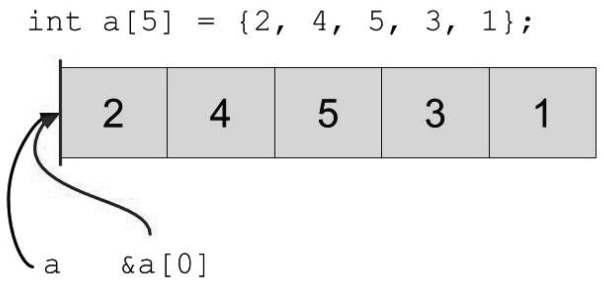
Arrays and Pointers

You might have already been able to make this observation, but arrays and pointers are related. The name of an array, much like that of a pointer, represents a memory address with the index representing the offset from that address. Unlike a pointer, the address to which an array refers cannot change.

- **Key Point:** Pointer variables are **variable** in that they can change the memory address to which they point. Arrays will always point to the same memory and are **constants** in that respect.

Note: The address of `a` and the address of `a[0]` are one and the same.

```
int a[5] = {2, 4, 5, 3, 1};

printf("Address of array a = %p\n", a);
printf("Address of array a[0] = %p\n", &a[0]);
```



In the example that follows the pointer notation is used in the declaration and the first line of the definition. Once inside the function the pointer variable can behave as if it were an array.

```
#define SIZE 5

void printArray(int*);

int main(void)
{
  int x[SIZE] = {3, 5, 7, 9, 10};

  printf("Array x address: %p\n", x);

  printArray(x);

  return(0);
}
```

```
void printArray(int *y)
{
  int i;

  printf("Array y address: %p\n", y);

  for(i = 0; i < SIZE; i++)
  {
    printf("y[%d] = %d\n", i, y[i]);
  }
}
```

What is true regarding the two memory addresses displayed in the example above?

This copyrighted text licensed to on via

Pointer Arithmetic and Arrays (Single Dimension Arrays only)

- What takes place when you attempt to add an integer to an address? The result is another address and this type of arithmetic can be used to access and change elements of an array.

```
#define SIZE 5

void printArray(int*);
void changeArray(int*);

int main(void)
{
  int x[SIZE] = {3, 5, 7, 9, 10};

  changeArray(x);
  printArray(x);

  return(0);
}
```

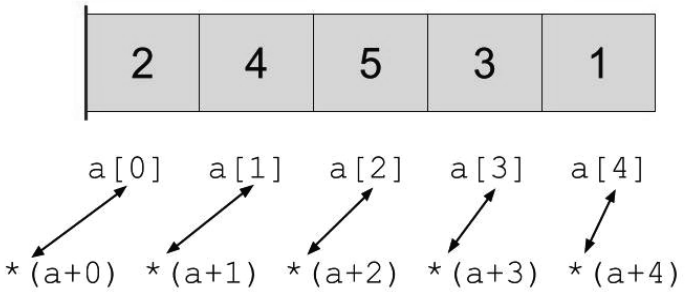
```
void changeArray(int *y)
{
  int i;
  int temp;

  temp = *y;

  for(i = 1; i < SIZE; i++)
  {
    *(y + i - 1) = *(y + i);
  }

  *(y + i - 1) = temp;
}
```

```
void printArray(int *y)
{
  int i;

  for(i = 0; i < SIZE; i++)
  {
    printf("y[%d] = %d\n", i, *(y + i));
  }
}
```

Output Generated:

```
y[0] = 5
y[1] = 7
y[2] = 9
y[3] = 10
y[4] = 3
```

```
int a[5] = {2, 4, 5, 3, 1};
```



The application of the **indirection operator (*)** to the sum of the starting memory address and an integer produces the element at the index of the integer.

```
a[x] == *(a + x)
```

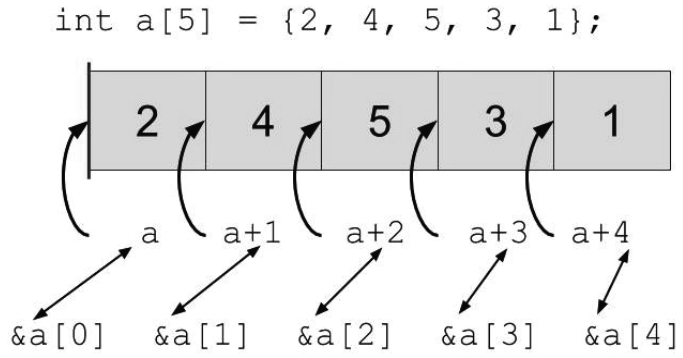

This copyrighted text licensed to on via

Want addresses?

As with index values, the value added to `a` represents an offset from the memory address value that `a` represents. The addition results in a new memory address. Don't different data types require different amounts of memory? Yes, but adding one to a pointer will advance to the next element in the array based on the type of the array. In the case of integer data on our server the addition of one will be an increment of four bytes.

This copyrighted text licensed to on via

```
void printArray(int*);

int main(void)
{
  int a[5] = {2, 4, 5, 3, 1};

  printf("Array a address: %p\n", a);

  printArray(a);

  return(0);
}

void printArray(int *y)
{
  int i;

  printf("Array y address: %p\n", y);

  for(i = 0; i < 5; i++)
  {
    printf("&y[%d] = %p\n", i, y + i);
  }
}
```



Output Generated:

```
Array a address: 0x7ffd003f7680
Array y address: 0x7ffd003f7680
&y[0] = 0x7ffd003f7680
&y[1] = 0x7ffd003f7684
&y[2] = 0x7ffd003f7688
&y[3] = 0x7ffd003f768c
&y[4] = 0x7ffd003f7690
```

Memory Allocation Functions

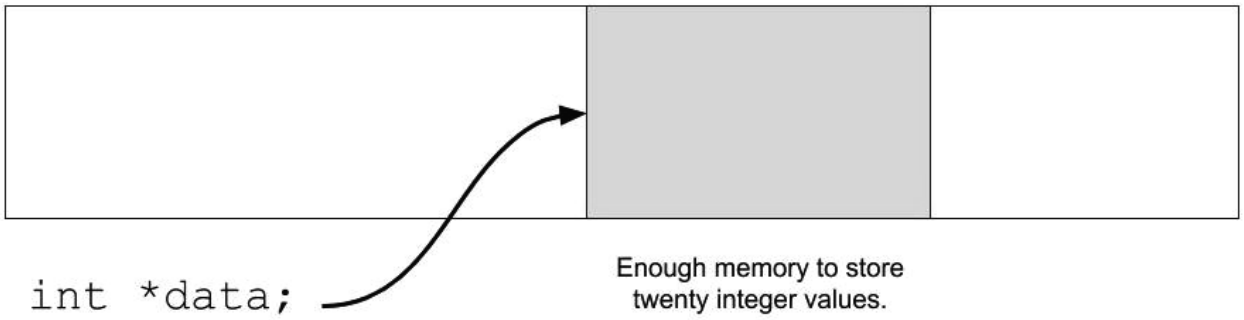
The declaration of the capacity of an array before the program is ever compiled has several limitations that you may have already considered. First, we may not reserve enough memory for the problem. Second, we may reserve too much limiting that which remains for other data. Our venture into dynamic memory allocation will improve our ability to request exactly the memory needed for the problem.

To dynamically allocate memory we will make use of a predefined function called `malloc` that is found in the `stdlib.h` library.

- The `malloc` (m-emory alloc-ation) function will allocate a specified number of bytes of memory. A pointer variable must be used to reference the starting address of the newly allocated memory returned from the `malloc` function.

How do we know how many bytes of memory to request? We may know the quantity of data but how to ensure the total is sufficient for the data type of the data set?

- The `sizeof` function will accept a data type and return the number of bytes needed to store one value of this type.
- The parameter to the `malloc` function will be the product of the value return from the `sizeof` function and the desired quantity of data.




```
data = (int*)malloc(sizeof(int) * 20);
```

Example - Ask the user for the quantity of integer data we need to store. Accept the data set as input and display it in the order entered.

```
int getQuantity(void);
void getData(int*, int);
void printData(int*, int);

int main(void)
{
  int *data;
  int quantity;

  quantity = getQuantity();

  data = (int*)malloc(sizeof(int) * quantity);
  getData(data, quantity);
  printData(data, quantity);

  return(0);
}
```

This copyrighted text licensed to on via

```
int getQuantity(void)
{
  int qty;

  printf("Enter quantity of data -> ");
  scanf("%d", &qty);

  return(qty);
}

void getData(int *data, int size)
{
  int i;

  printf("Enter %d integers -> ", size);

  for(i = 0; i < size; i++)
  {
    scanf("%d", data + i);
  }
}

void printData(int *data, int size)
{
  int i;

  printf("\nContents of data set: ");

  for(i = 0; i < size; i++)
  {
    printf("%d ", *(data + i));
  }

  printf("\n");
}
```

Note: for the sake of simplicity we are skipping the important tasks of confirming that allocation was successful and releasing memory when finished.

This copyrighted text licensed to on via

Chapter 11 – Strings: Character Arrays

- A **string** is a series of characters with the capability to be treated as a single unit.

The C programming language does not have a string data type but attempts to provide some capacity to work with string data through the use of character arrays.

Similar to numeric arrays we can declare a string to be of any predetermined size. A problem common to character arrays involves the number of elements necessary for the string data can vary during the execution of the program. For example, a given character array may have a capacity of 20 elements and initially stores "Computer" with that data being replaced later with "Science". How do we, and the functions that we using with string data, know where the data ends and the unused portion of the array begins?

We are able to identify where the data in a character array ends through the use of a character known as a delimiter. This delimiter character `'\0'` represents the end of the data within a character array. This will be important to many functions that operate with strings.

Strings in the C programming language are **variable-length** and **delimited** strings.

- What is variable-length? The amount of data found in the array at a given time.
- What does it mean to be delimited? A character is inserted into the array to separate the data from the unused portion of the array.

Declaring Strings and Accessing String Elements

- We are able to access each individual elements of a character array in a manner similar to that of numeric arrays.

Examining the elements of a character array:

```
int lcv;
char str[11] = "Good Day";

for(lcv = 0; lcv < 11; lcv++)
{
  printf("str[%2d] = %3c %5d\n", lcv, str[lcv], str[lcv]);
}

printf("\n");
```

- Where is the delimiter? Is it there? Yes! The ASCII value of the delimiter is zero and is found properly placed at index 8. The initialization statement will add the delimiter to the end of the character array.

Output:

```
str[ 0] =   G    71
str[ 1] =   o   111
str[ 2] =   o   111
str[ 3] =   d   100
str[ 4] =        32
str[ 5] =   D    68
str[ 6] =   a    97
str[ 7] =   y   121
str[ 8] =        0
str[ 9] =        0
str[10] =        0
```

What is in the remaining **unused elements** (at index 9 and 10) of the character array?

- Because we initialized the array above the remaining elements are filled with zeros.

This copyrighted text licensed to on via

String Input/Output Functions

Here is a point to ponder, if we want to load data into an array must we always input that data one element at a time until the array is filled? Would this imply that we have to enter one character at a time in order to input a string?

- Good news! The answer is no! While we do not have a string data type available we do have many functions and features in the C programming language that allows us to operate with strings with some convenience.

The `%s` Placeholder

- The `%s` placeholder represents a string and is used in both familiar input and output functions.

Example of `%s` placeholder in action:

```
char str1[SIZE];

printf("Enter a string -> ");
scanf("%s", str1);           //NOTICE NO &, WHY IS THIS?!

printf("The string you entered was %s.\n", str1);
```

Output of previous code segment

```
Enter a string -> Purdue
The string you entered was Purdue.
```

- The `scanf` function only reads in the data up until a new line (`'\n'`) or any white space is entered.

```
Enter a string -> Go Purdue
The string you entered was Go.
```

The solution? Find an alternative to `scanf`!

```
printf("Enter a string -> ");
gets(str1);

printf("The string you entered was %s.\n", str1);
```

What happens to the `'\n'` character entered by the user? The `gets` function will accept the input, terminated by a new line, and make a delimiter-terminated string out of it.

This copyrighted text licensed to on via

Concerns with both `scanf` and `gets` functions:

- The compiler we are using this semester no longer remains silent regarding the potential problems associated with the `gets` function. `warning: the `gets' function is dangerous and should not be used.`

What makes `gets` (and for the same reason `scanf`) "dangerous" and "should not be used"?

- Neither function will stop accepting input when the character array has been filled to its defined capacity.
- Exceeding the defined capacity of the array will result in the same problems associated with range checking issues.

Alternative (safer) approach to input:

```
void getString(char x[])
{
  int index = 0;

  printf("Enter your string -> ");

  do
  {
    x[index++] = getchar();
  }while(index < SIZE && x[index - 1] != '\n');

  //REPLACE THE FINAL CHARACTER WITH A DELIMITER CHARACTER
  x[index - 1] = '\0';
}
```

- The `getchar` function will accept the input of a single character.
- The `do-while` above will terminate the accepting of input **when the user enters a new line character** or **when the defined size of the array has been reached**.
- Because the `getchar` function handles only individual character input it does not place the delimiter character in the array, but it will place the new line character (ASCII value 10) into the array, the delimiter must be assigned to the array and will overwrite the new line character in the function above.

String Manipulation Functions

The C programming language provides us with an adequate set of string functions that can be found in the `string.h` library. Functions we will consider include: `strcpy`, `strcmp`, `strlen`, and we will come back later to see an example using `strstr`. Similar to `printf` and `scanf` these functions will use the addresses represented by the names of the arrays and the delimiters stored within to complete their tasks.

This copyrighted text licensed to on via

Determining the Length of a String

The first function from `string.h` to introduce is the `strlen` function which will return the number of characters in the string. This function does not count the terminal delimiter character, but the value returned does represent the location (index) of the delimiter in the array.

```
char str1[SIZE];
int length;

getString(str1);
length = strlen(str1);

printf("Your string [length %d]: %s\n", length, str1);
```

Example Executions:

`Enter your string -> Purdue` `Your string [length 6]: Purdue`	`Enter your string -> Go Purdue!1!!` `Your string [length 13]: Go Purdue!1!!`

Copying Strings, it is easy to make a copy of a single variable, but how about strings? Can we use the assignment operator?

```
char str1[SIZE];
char str2[SIZE];

getString(str1);  //SEE USER-DEFINED FUNCTION ON PREVIOUS PAGE.

str2 = str1;

printf("The string you entered was %s.\n", str1);
printf("The copy of the string you entered is %s.\n", str2);
```

What is the message generated by the compiler?

- `error: incompatible types when assigning to type 'char[150]' from type 'char *'`

What is REALLY being assigned in the expression `str2 = str1` found in the example above?

- The assignment is an attempt to force `str2` to abandon the memory to which it was allocated and to refer to the same memory as `str1`. If this were possible there would still be only a single copy of the string in memory.

This copyrighted text licensed to on via

Why does the compiler not approve of this particular assignment expression?

- The memory allocated to a statically declared array is fixed in terms of size and location.
- Based on the previous statement we would say the memory address to which an array refers is **constant**.

Alternative - using the `string.h` function `strcpy`

- The function `strcpy` will take the **second** argument and copy its value to the **first**. In this case, the contents of `str1` are copied to `str2`.

```
char str1[SIZE];
char str2[SIZE];

getString(str1);

strcpy(str2, str1);

printf("Your input: %s\n", str1);
printf("Your input copy: %s\n", str2);
```

Output:

```
Enter your string -> Go Purdue!
Your input: Go Purdue!
Your input copy: Go Purdue!
```

Is there a concern with more data being copied than there were spaces reserved in memory?

- Yes! The `strcpy` function will continue to copy characters until the delimiter is found. Verify that the destination array has the needed capacity!

This copyrighted text licensed to on via

Comparing Two Strings

- Just as we cannot assign one string to another, neither can we successfully compare two strings for equality using the equality and relational operators.

Example:

```
char str1[SIZE];
char str2[SIZE];

getString(str1);

strcpy(str2, str1);

if(str1 == str2)
{
  printf("The strings are equal!\n");
}
```

One would assume if the equality operator compares contents of strings that ANY value entered would lead to the equal statement being printed every time the previous code segment is executed.

What is output? None.

What does that tell us about what is being compared when using relational or equality operators with two strings?

The comparison using the equality operator is testing memory addresses and not the string data.

Alternative - using the `string.h` function `strcmp`

This third function from the `string.h` library will accept two strings as parameters and will return an integer indicating how the two strings are different. Two strings storing the same data have no difference.

`Enter your string -> company`
`Enter your string -> corporate`
`Compare result: -5`
	In the first example it is the third element where the first difference is found. This difference value is returned from the `strcmp` function. The element of the second parameter is subtracted from the the element of the first parameter.

`Enter your string -> company`
`Enter your string -> companion`
`Compare result: 16`
	`Enter your string -> ABC123`
	`Enter your string -> ABC123`
	`Compare result: 0`

`Enter your string -> AB1C23`
`Enter your string -> ABC123`
`Compare result: -18`
	It is NOT the sum of the differences that is used when calculating the difference, it is the FIRST difference in order from lowest index to highest.

`Enter your string -> ABC12`
`Enter your string -> ABC123`
`Compare result: -51`
	What if the two strings do not have the same number of elements?

`Enter your string -> ABC123`
`Enter your string -> AbC123`
`Compare result: -32`
	Does character case have significance?

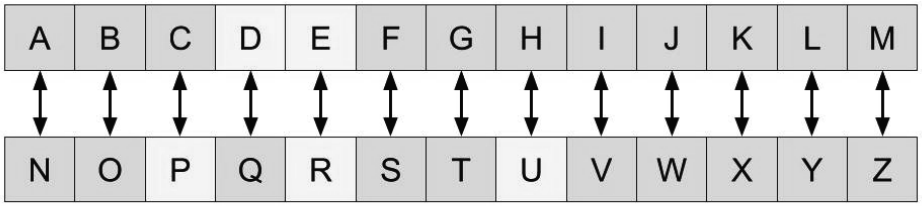
This copyrighted text licensed to on via

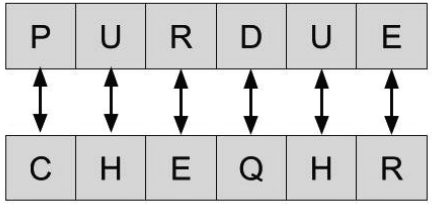
The code:

```
char str1[SIZE];
char str2[SIZE];
int compare;

getString(str1);
getString(str2);

compare = strcmp(str1, str2);

printf("Compare result: %d\n", compare);
```

Problem: Implement the ROT13 encryption algorithm. ROT13 is not really an encryption algorithm but processing text through this method does make it challenging to read. To those characters in the first half of the alphabet we add 13 to rotate them into the second half. And from those characters in the second half of the alphabet we subtract 13 to rotate them into the first half. Characters that are not alpha characters will not be changed in the process. It is important to note that the process of encryption is the same as decryption.

```
Enter your string -> Go Purdue!!!
Original String: Go Purdue!!!
Rotated String: Tb Cheqhr!!!
Original String: Go Purdue!!!
```





This copyrighted text licensed to on via

Solution:

```
#define SIZE 51

void getString(char[]);
void rot13(char[]);

int main(void)
{
  char str1[SIZE];

  getString(str1);

  printf("Original String: %s\n", str1);
  rot13(str1);
  printf("Rotated String: %s\n", str1);
  rot13(str1);
  printf("Original String: %s\n", str1);

  return(0);
}

void rot13(char x[])
{
  int lcv;

  for(lcv = 0; lcv < strlen(x); lcv++)
  {
    if((x[lcv] >= 'a' && x[lcv] <= 'z') || (x[lcv] >= 'A' && x[lcv] <= 'Z'))
    {
      if((x[lcv] >= 'a' && x[lcv] <= 'm') || x[lcv] <= 'M')
      {
        x[lcv] += 13;
      }
      else
      {
        x[lcv] -= 13;
      }
    }
  }
}
```

This copyrighted text licensed to on via

Searching for a Sub-string

The final function (`strstr`) to introduce from the `string.h` library has the capacity to locate a sub-string within string.

The `strstr` function accepts two memory addresses as parameters and returns the location of the sub-string within the string. When the sub-string cannot be found the value returned is a defined zero known as NULL.

If the function returns an address then to which type of variable would this value be assigned? Answer: a character pointer.

Problem - given a string and sub-string determine the number of times the sub-string appears in the string.

```
Enter the string -> ATATATA
Enter the substring -> AT
The substring AT appears 3 time(s) in the string ATATATA.

Enter the string -> ATATATA
Enter the substring -> ATA
The substring ATA appears 3 time(s) in the string ATATATA.
```

Solution:

```
#define SUB_INPUT 1
#define STR_INPUT 2

#define SUBSTR 21
#define STRING 50

void getString(char[], int, int);
int ctStrings(char[], char[]);

int main(void)
{
  char sub[SUBSTR];
  char str[STRING];
  int count;

  getString(str, STR_INPUT, STRING);
  getString(sub, SUB_INPUT, SUBSTR);
  count = ctStrings(str, sub);

  printf("The substring %s appears %d time(s) in the string %s.\n", sub, count, str);

  return(0);
}
```

This copyrighted text licensed to on via

```
void getString(char x[], int option, int size)
{
  int index = 0;

  if(option == SUB_INPUT)
  {
    printf("Enter the substring -> ");
  }
  else if(option == STR_INPUT)
  {
    printf("Enter the string -> ");
  }

  do
  {
    x[index++] = getchar();
  }while(index < size && x[index - 1] != '\n');

  x[index - 1] = '\0';
}

int ctStrings(char str[], char sub[])
{
  int ct = 0;
  char *ptr;

  ptr = strstr(str, sub);  //LOOK FOR sub AT START OF str

  while(ptr != NULL)
  {
    ct++;
    ptr = strstr(ptr + 1, sub); //PERMITS OVERLAP
    //ptr = strstr(ptr + strlen(sub), sub); //NO OVERLAP
  }

  return(ct);
}
```

This copyrighted text licensed to on via

Fall 2025 Exams and Solutions

Exam #1:

1. B	2. C	3. C	4. A	5. B
6. C	7. C	8. A	9. B	10. A
11. A	12. B	13. C	14. A	15. C
16. D	17. B	18. C	19. D	20. D
21. C	22. B	23. C	24. B	25. A
26. A	27. B	28. C	29. C	30. A
31. C	32. A	33. B	34. C	35. D

Exam #2:

1. B	2. A	3. C	4. B	5. A
6. B	7. A	8. A	9. C	10. B
11. D	12. A	13. C	14. A	15. B
16. D	17. B	18. C	19. B	20. A
21. C	22. B	23. B	24. A	25. C
26. A	27. B	28. C	29. B	30. D
31. B	32. B	33. B	34. D	35. D

Final Exam:

1. C	2. A	3. D	4. B	5. C
6. C	7. A	8. A	9. B	10. A
11. C	12. C	13. C	14. D	15. D
16. C	17. A	18. D	19. C	20. B
21. B	22. A	23. B	24. A	25. A
26. C	27. A	28. C	29. C	30. C
31. B	32. B	33. B	34. C	35. A
36. A	37. C	38. B	39. A	40. A
41. B	42. C	43. C	44. C	45. A
46. C	47. C	48. A	49. C	50. A

How are old exams best used in the process of preparing for an exam?

- At the end of your studying you should attempt an old exam and do so at a location on campus that will allow you to simulate the exam environment. Take the exam, grade your work, and identify what subjects or types of problems need further review prior to the exam date.

Where can I find more problems for practice?

- Boilerexams.com
- See the problems at the end of each chapter in the C programming text. A link to the publisher's site which includes student downloads and solutions is available on Brightspace.

Why is some of the code so complex in these old exams?

- Every exam is specific to the semester it was given. This not only includes the topics covered based on when the exam took place, but also the assignments that have been given in the course that semester.
- Many code segments are directly from previous homework and lab programming concepts and those students would have some level of familiarity with the problem and logic being presented. This would make interpreting the code and the logic it represents easier and faster.

This copyrighted text licensed to on via

This copyrighted text licensed to on via

Fall 2025 CS 15900 Midterm 1
Tuesday October 7, 2025 (8:00 – 9:00 PM)

35 problems × 3 points each = 105 points
Any points earned beyond 100 are considered extra credit

Exam Rules/Notices:

1. Complete your name and your student identification number on the answer sheet (write in AND bubble-in all requested information).
    - The *test form* and *section* fields can be left blank.
    - **An incorrect student ID number will result in a zero for this exam.** All student ID numbers begin with `00` (zero-zero) for a total of **TEN** digits. This must be written and bubbled-in on your exam answer sheet.
    - **You must be in your assigned seat to earn credit for this exam.**
2. Make sure you have all 35 problems for this exam. You should answer every problem. All problems only have one best answer. Please be careful when filling in your answers. Any answer determined to be unreadable by Instructional Data Processing will be considered incorrect. Only the answer on your answer sheet can be considered when grading.
    - An **operator precedence and ASCII table** have been provided on the last page of this exam form.
3. **NO QUESTIONS WILL BE ANSWERED DURING THE EXAM.** We do not answer questions during the exam due to the limited number of staff members present. It is easier for us to compensate for an erroneous test problem than it is to answer questions or to effectively make an announcement during the examination period. **Manage your time accordingly!**
4. You must assume that all C programs and code segments are implemented in Vocareum and would be compiled with the `gcc` compiler. When only segments of code are presented, you may assume that the appropriate `#include` statements are present and that the code would be inside of a fully operational function or program.
5. **Protect your work and keep your focus on your own exam.** It is considered dishonest if you copy from another student or to permit another student to copy from you. Anyone found to violate course policies will receive a **failing grade for the course** and will be referred to the Office of the Dean of Students.
6. **You are NOT permitted to use any resources during this exam.** This includes but is not limited to texts, notes, calculators, cell phones, audio devices, and computers. If any personal electronic device is audible during the exam, then you will be required to immediately submit your exam and leave the testing facility.
7. **When you are finished with the exam, please proceed to the lobby** to submit your answer sheet and to exit the facility. You will be required to show photo identification before we can accept your work. You will keep this exam form. You may not return to your seat once you have submitted your exam.
8. **You may anticipate that scores and official exam answers will appear on Brightspace by 5:00 PM on Friday October 10.** Do not contact course staff members about mistakes (if any) until answers are posted. If you feel an error remains after answers have been posted, then you may make a written request for a regrade by Friday October 17 and deliver it to your lecturer during their office hours. Your request must be a hard copy (no email) and include a detailed description, using code if applicable, demonstrating your belief why the posted answer is incorrect.
9. **When time is called ALL writing must stop**. You are not permitted to continue to revise any answer on your exam once time has been called.
10. **Schedule update.**
    - Labs will continue to meet this week (Tuesday October 7 – Friday October 10) for Lab 6.
    - Lecture WILL meet this Thursday, October 9. Lecture will be canceled on Tuesday December 9 in compensation for this exam.

NO QUESTIONS WILL BE ANSWERED DURING THE EXAM.

This copyrighted text licensed to on via

Use the code segment below for problem #1

```
int x = 13;
int y = 12;
int z = 19;
int result;

result = z % y + y % x;
result += (1 - x / y) * x + (1 - y / x) * y;
result -= z / (x / y);

printf("result: %d\n", result);
```

1. Which of the following is the output generated by the code segment above?
   A. `result: 0`
   B. `result: 12`
   C. `result: 13`
   D. None of the above.

Use the code segment below for problems #2 - #4

```
int a = 4;
int b = 7;
float c = 3.5;

a = a + b * c - 5;
b = b / (a % 2 / 2.2 * c);
c = c * ((++a - b++) % 3);

printf("a: %d b: %d\n", a, b);
printf("c: %d\n", (int) c);
```

2. Which of the following is the first line of output generated by the code segment above?
   A. `a: 23 b: 4`
   B. `a: 33 b: 4`
   C. `a: 24 b: 5`
   D. None of the above.

3. Which of the following is the second line of output generated by the code segment above?
   A. `c: 0`
   B. `c: 3.5`
   C. `c: 7`
   D. None of the above.

4. If included in the code segment above, which of the following statements would result in a compile-time error?
   A. `b = (float) b / (a % (2 / 2.2) * c);`
   B. `b = b / (a % 2 / (int) (2.2 * c));`
   C. `b = b / (a % (int) (2 / 2.2) * c);`
   D. None of the above.

This copyrighted text licensed to on via

Use the program below for problems #5 - #6

```
#include<stdio.h>

int makeCalculation(int, int);

int main()
{
  int a = 11;
  int b = 40;
  int c = 25;

  c += makeCalculation(a++, b++);
  a += makeCalculation(++c, b++);
  b += makeCalculation(++a, ++c);

  printf("a: %d\n", a);
  printf("b: %d c: %d\n", b, c);

  return(0);
}

int makeCalculation(int x, int y)
{
  int result;

  result = y - (y / x) * x;

  return(result);
}
```

5. Which of the following is the first line of output generated by the program above?
   A. `a: 20`
   B. `a: 21`
   C. `a: 19`
   D. None of the above.

6. Which of the following is the second line of output generated by the program above?
   A. `b: 57 c: 32`
   B. `b: 56 c: 33`
   C. `b: 55 c: 34`
   D. None of the above.

7. Which of the following statements about passing parameters in C functions is FALSE?
   A. In the passing by value technique only a copy of the data is sent from the calling function to the called function.
   B. Given the address of a variable from the calling function, the called function can access and manipulate the value stored in that memory location.
   C. When designing a function the parameters being passed by address must be listed before those being passed by value.
   D. None of the above.

This copyrighted text licensed to on via

Use the program below for problems #8 - #9

```
#include<stdio.h>

int square(int);

int main()
{
  int a = 3;
  int b;

  b = square(square(a++));

  printf("first b: %d\n", b);

  b = square(++a + square(1));

  printf("second a: %d b: %d\n", a, b);

  return 0;
}

int square(int x)
{
  int y;

  y = x++;

  return(y * x);
}
```

8. Which of the following is the first line of output generated by the program above?
   A. `first b: 156`
   B. `first b: 144`
   C. `first b: 81`
   D. None of the above.

9. Which of the following is the second line of output generated by the program above?
   A. `second a: 5 b: 25`
   B. `second a: 5 b: 56`
   C. `second a: 5 b: 49`
   D. None of the above.

10. Which of the following statements regarding the scope of variables in C is TRUE?
    A. A variable declared in the local declaration section of a function has a scope that extends until the end of that function.
    B. It is good programming style to reuse identifiers within the same or overlapping scope.
    C. The scope of a variable is unrelated to the location of the program in which it is declared.
    D. None of the above.

This copyrighted text licensed to on via

Use the program below for problems #11 - #12

```
#include<stdio.h>
#include<math.h>

#define PI 3.1415

int calcPrec(float, int);

int main()
{
  int z;

  z = calcPrec(PI, 3);

  printf("Result 1: %.*f\n", z, PI);

  z = calcPrec(2 * PI, 2);

  printf("-=-=-=-=-=-=-=-=-=-\n");
  printf("Result 2: %*.3f\n", z, PI);

  return(0);

}

int calcPrec(float x, int y)
{
  return((int)(x * pow(10, y)) % 10);
}
```

11. Which of the following is the first line of output generated by the program above?
    A. `Result 1: 3.1`
    B. `Result 1: 3.142`
    C. `Result 1: 3.14`
    D. None of the above.

12. Which of the following is the final **two** lines of output generated by the program above?

A	`-=-=-=-=-=-=-=-=-` `Result 2:  3.142`	C	`-=-=-=-=-=-=-=-=-` `Result 2:  3.142`
B	`-=-=-=-=-=-=-=-=-` `Result 2:  3.142`	D	None of the above.

13. Which of the following statements regarding tasks factored into user-defined functions is TRUE?
    A. A function that performs several unrelated tasks is functionally cohesive.
    B. It is good design practice to repeat the logic of one task in multiple functions of the program.
    C. A function that does one and only one task of the larger problem is functionally cohesive.
    D. None of the above.

This copyrighted text licensed to on via

Use the program below for problems #14 - #16

```
#include<stdio.h>

int gradeCheck(int, int);

int main()
{
  int pts = /* See problems 14 and 15 */;
  int gpa;
  char grade = 'F';

  gpa = gradeCheck(pts, 165);
  gpa += gradeCheck(pts, 195);
  gpa += gradeCheck(pts, 225);
  gpa += gradeCheck(pts, 255);

  grade -= gpa;

  printf("Grade earned: %c\n", grade);

  return(0);
}

int gradeCheck(int total, int min)
{
  total = total / min;

  return((total + 2) % (total + 1));
}
```

14. Which of the following is the output generated if the `pts` variable were initialized to `210`?
    A. `Grade earned: D`
    B. `Grade earned: C`
    C. `Grade earned: E`
    D. None of the above.

15. Which of the following is the output generated if the `pts` variable were initialized to `0`?
    A. `Grade earned: E`
    B. `Grade earned: D`
    C. `Grade earned: F`
    D. None of the above.

16. Which of the following initial values of the `pts` variable would generate an output grade of `'A'`?
    A. 255
    B. 330
    C. 500
    D. None of the above.

17. Which of the following statements regarding local variables in C is TRUE?
    A. A variable declared in the local declaration section of a function can have the same identifier as one of the parameters of the same function.
    B. Parameters are defined as local variables in the first line of the function definition and should not be re-defined within the local declaration section of the same function.
    C. A local variable can be directly referenced outside of the function in which it is defined using its identifier.
    D. None of the above.

This copyrighted text licensed to on via

Use the program below for problems #18 - #19

```
#include<stdio.h>

int valueChanges(int, int*);

int main()
{
  int x;
  int y;

  printf("Enter two integers -> ");
  scanf("%d %d", &x, &y);

  x = valueChanges(x, &y);

  printf("x: %d y: %d\n", x, y);

  return(0);
}

int valueChanges(int x, int *ay)
{
  x = x - *ay;
  *ay = *ay + x;
  x = *ay - x;

  return(x);
}
```

18. Which of the following is the output generated if given integers `11 5` as input?
    A. `x: 5 y: 5`
    B. `x: 11 y: 11`
    C. `x: 5 y: 11`
    D. None of the above.

19. Which pair of integers given as input will generate `x: -4 y: 3` as output?
    A. `-3 4`
    B. `4 -3`
    C. `-3 -4`
    D. None of the above.

20. Which of the following statements regarding the evolving role of the `main` function correctly aligns with the standards of the course?
    A. All declarations of variables and functions, along with most, if not all calls to use-defined functions, are to appear within the `main` function of a program.
    B. The role of the `main` function is to coordinate the function calls, to establish the data needs, and generate output for a program.
    C. It is expected that the `main` function be the location of calls to user-defined functions that contain the critical tasks of a program while at the same time completing trivial tasks for which a new user-defined function would be a burden.
    D. None of the above.

This copyrighted text licensed to on via

Use the program below for problems #21 - #22

```
#include<stdio.h>

#define HEIGHT 3.5
#define FRACTION 4 / 5

int main(void)
{
  float base1;
  float area_denom;
  float base2;

  base1 = 1.1 * FRACTION + FRACTION + HEIGHT;
  base2 = 2.3 * HEIGHT / FRACTION + 4;

  area_denom = HEIGHT * 2 + (float) FRACTION;

  printf("Base1: %.1f Base2: %.1f\n", base1, base2);
  printf("Area denominator: %.1f\n", area_denom);

  return 0;
}
```

21. Which of the following is the first line of output generated by the program above?
    A. `Base1: 3.5 Base2: 14.06`
    B. `Base1: 5.2 Base2: 0.0`
    C. `Base1: 4.4 Base2: 4.4`
    D. None of the above.

22. Which of the following is the second line of output generated by the program above?
    A. `Area denominator: 7.0`
    B. `Area denominator: 7.8`
    C. `Area denominator: 2.2`
    D. None of the above.

23. Which of the following statements about programming languages is TRUE?
    A. Only the machine and symbolic languages can be understood by computer hardware.
    B. The C programming language is not considered a high-level programming language because of its age.
    C. High-level languages are portable to many different computers.
    D. None of the above.

24. Which of the following statements about the waterfall model of software development is TRUE?
    A. The implementation of code is the first step in the waterfall model of developing software.
    B. The waterfall model of developing software places an emphasis on designing and testing.
    C. All steps of the waterfall model are active simultaneously as a programming project moves from start through completion.
    D. None of the above.

25. Which of the following expressions will generate a random number from 1 to 10 inclusive of both endpoints?
    A. `rand() % 10 + 1`
    B. `rand(1, 10)`
    C. `(rand() + 1) % 10`
    D. None of the above.

This copyrighted text licensed to on via

Use the program below for problems #26 - #27

```
#include<stdio.h>

int convert(int, int*);

int main()
{
  int start = 200000;
  int unitA;
  int unitB;
  int unitC;
  int unitD;

  unitA = convert(60 * 60 * 24, &start);
  unitB = convert(60 * 60, &start);
  unitC = convert(60, &start);
  unitD = convert(1, &start);

  printf("unitA: %d unitB: %d\n", unitA, unitB);
  printf("unitC: %d unitD: %d\n", unitC, unitD);

  return(0);
}

int convert(int factor, int *ct)
{
  int value;

  value = *ct / factor;
  *ct %= factor;

  return(value);
}
```

26. Which of the following is the first line of output generated by the program above?
    A. `unitA: 2 unitB: 7`
    B. `unitA: 3 unitB: 8`
    C. `unitA: 4 unitB: 6`
    D. None of the above.

27. Which of the following is the second line of output generated by the program above?
    A. `unitC: 34 unitD: 0`
    B. `unitC: 33 unitD: 20`
    C. `unitC: 35 unitD: 10`
    D. None of the above.

28. Which of the following statements regarding the rules for the selection of an identifier is FALSE?
    A. In the C programming language an identifier cannot begin with a digit character.
    B. The standards of the course require a descriptive name for every declared variable and function.
    C. Only the first 31 characters of an identifier are meaningful in the C programming language.
    D. None of the above.

This copyrighted text licensed to on via

Use the code segment below for problems #29 - #30

```
int x = 10;
int y = 9;

float a = 1.5;
float b = 2.5;

x = x / (int) b;

a = (float) y / a;

printf("total 1: %.1f\n", x + a);

y = y / b;

b *= 5;

printf("total 2: %.1f\n", y + b);
```

29. Which of the following is the first line of output generated by the code segment above?
    A. `total 1: 9.0`
    B. `total 1: 10.0`
    C. `total 1: 11.0`
    D. None of the above.

30. Which of the following is the second line of output generated by the code segment above?
    A. `total 2: 15.5`
    B. `total 2: 13.0`
    C. `total 2: 16.1`
    D. None of the above.

31. Which of the following statements regarding an assignment type conversion is FALSE?
    A. An assignment conversion occurs when the data type of a parameter in a function call does not match the data type of the corresponding parameters in the function definition.
    B. When the variable on the left is of a higher ranked data type than the result of the expression on the right, the assignment conversion that occurs will be a data type promotion.
    C. The result of assigning a value of a higher ranked type to a variable of a lower ranked type is undefined in the C programming language.
    D. None of the above.

32. Which of the following statements regarding variables in the C programming language is TRUE?
    A. The declared data type of a variable cannot change during the execution of a program.
    B. Locally-declared variables are initialized automatically by the compiler.
    C. All operators can be applied to any variable.
    D. None of the above.

33. Which of the following is being described when a function is referred to by its data type?
    A. The type of parameters being received by the function.
    B. The type of value being returned from the function.
    C. Both the data type of the parameters received and the type of value being returned.
    D. None of the above.

This copyrighted text licensed to on via

34. Which of the following statements regarding placeholders in `printf` and `scanf` is FALSE?
    A. Both the `long` data type and `double` data type use the same size code in a placeholder.
    B. A flag of zero in the placeholder will specify padding with the digit zero instead of the space character.
    C. When the left alignment of output is needed the use of the + character is included in the placeholder.
    D. None of the above.

35. Which of the following statements regarding the class schedule is TRUE?
    A. All lecture sections will meet as scheduled and include a lecture quiz on Thursday October 9.
    B. All regularly scheduled lab sections will meet October 7 – October 10 for Lab 6.
    C. Lecture will be canceled on Tuesday December 9 in compensation for this exam.
    D. All of the above!

This copyrighted text licensed to on via

This copyrighted text licensed to on via

Fall 2025 CS 15900 Midterm 2
Monday November 10, 2025 (8:00 – 9:30 PM)

35 problems × 3 points each = 105 points
Any points earned beyond 100 are considered extra credit

Exam Rules/Notices:

1. Complete your name and your student identification number on the answer sheet (write in AND bubble-in all requested information). The *test form* and *section* fields can be left blank.
    - **An incorrect student ID number will result in a zero for this exam.** All student ID numbers begin with `00` (zero-zero) for a total of **TEN** digits. This must be written and bubbled-in on your exam answer sheet.
    - **You must be in your assigned seat to earn credit for this exam.**
2. Make sure you have all 35 problems for this exam. You should answer every problem. All problems only have one best answer. Please be careful when filling in your answers. Any answer determined to be unreadable by Instructional Data Processing will be considered incorrect. Only the answer on your answer sheet can be considered when grading.
    - An **operator precedence and ASCII table** have been provided on the last page of this exam form.
3. **NO QUESTIONS WILL BE ANSWERED DURING THE EXAM.** We do not answer questions during the exam due to the limited number of staff members present. It is easier for us to compensate for an erroneous test problem than it is to answer questions or to effectively make an announcement during the examination period. **Manage your time accordingly!**
4. You must assume that all C programs and code segments are implemented in Vocareum and would be compiled with the `gcc` compiler. When only segments of code are presented, you may assume that the appropriate `#include` statements are present and that the code would be inside of a fully operational function or program.
5. **Protect your work and keep your focus on your own exam.** It is considered dishonest if you copy from another student or to permit another student to copy from you. Anyone found to violate course policies will receive a **failing grade for the course** and will be referred to the Office of the Dean of Students.
6. **You are NOT permitted to use any resources during this exam.** This includes but is not limited to texts, notes, calculators, cell phones, audio devices, and computers. If any personal electronic device is audible during the exam, then you will be required to immediately submit your exam and leave the testing facility.
7. **When you are finished with the exam, please proceed to the lobby** to submit your answer sheet and to exit the facility. You will be required to show photo identification before we can accept your work. You will keep this exam form. You may not return to your seat once you have submitted your exam.
8. **You may anticipate that scores and official exam answers will appear on Brightspace by 5:00 PM on Monday November 17.** Do not contact course staff members about mistakes (if any) until answers are posted. If you feel an error remains after answers have been posted, then you may make a written request for a regrade by Monday November 24 and deliver it to your lecturer during their office hours. Your request must be a hard copy (no email) and include a detailed description, using code if applicable, demonstrating your belief why the posted answer is incorrect.
9. **When time is called ALL writing must stop**. You are not permitted to continue to revise any answer on your exam once time has been called.
10. **Schedule notice.**
    - Labs will continue to meet this week (Tuesday November 11 – Friday November 14) for Lab 10.
    - Lecture **WILL** meet tomorrow Tuesday, November 11 and on Thursday, November 13.
    - Lecture will be canceled on Thursday **December** 11 in compensation for this exam.

NO QUESTIONS WILL BE ANSWERED DURING THE EXAM.

This copyrighted text licensed to on via

Use the code segment below for problems 1 – 2

```
int a = 5;
int b = 7;
int c = 9;

if (a > b + c)
{
  a = b++;
}
if (c < b + ++a)
{
  c = b + a;
}
else
{
  b = a + b + c;
}

printf("a = %d, b = %d, c = %d\n", a, b, c);
printf("Result: %d\n", a + b > c || c > a);
```

1. What is the FIRST line of output generated by the code segment above?

   A. `a = 5, b = 8, c = 9`
   B. `a = 6, b = 7, c = 13`
   C. `a = 6, b = 8, c = 13`
   D. None of the above.

2. What is the SECOND line of output generated by the code segment above?

   A. `Result: 1`
   B. `Result: 0`
   C. `Result: 9`
   D. None of the above.

3. Which of the following statements about short-circuit evaluation is TRUE?

   A. Short-circuiting applies only to logical expressions in `if` statements.
   B. Short-circuiting does not apply to logical expressions serving as loop control expressions.
   C. Short-circuiting applies to any logical expression including any of the binary logical operators.
   D. None of the above.

4. Which of the following statements about looping constructs is TRUE?

   A. The `while` loop always iterates at least one time, even if the initial evaluation of the loop control expression is false.
   B. The `do-while` loop iterates at least one before the initial evaluation of the loop control expression, making it useful when actions must occur prior to evaluating continuation.
   C. The `for` loop is a post-test loop that iterates before the initial evaluation of the loop control expression.
   D. None of the above.

This copyrighted text licensed to on via

Use the code segment below for problems 5 – 6

```
int a = 4;
int b = 6;
int c = 8;

if (b % 2 && a++)
{
  c = c - a;
}
else if (a == b || c--)
{
  b = c + a;
}
else
{
  a = a + b - c;
}

printf("a = %d, b = %d, c = %d\n", a, b, c);
printf("Result: %d\n", !(b % a));
```

5. What is the FIRST line of output generated by the code segment above?

   A. `a = 4, b = 11, c = 7`
   B. `a = 5, b = 11, c = 7`
   C. `a = 4, b = 10, c = 8`
   D. None of the above.

6. What is the SECOND line of output generated by the code segment above?

   A. `Result: 1`
   B. `Result: 0`
   C. `Result: -1`
   D. None of the above.

7. Which of the following statements regarding course standards when it comes to input validation is TRUE?

   A. The use of input validation ensures meaningful values are obtained before the input function is permitted to terminate.
   B. Use pass by address is advisable in most input functions utilizing validation logic.
   C. Input validation is used to ensure that the data type of the value given matches the expected data type.
   D. None of the above.

8. Which of the following statements regarding counter-controlled processes is TRUE?

   A. You can determine the total number of iterations before evaluating the loop control expression for the first time in a counter-controlled process.
   B. The number of iterations in a counter-controlled process depends on an unpredictable event.
   C. A counter-controlled process must be implemented using a pretest looping construct.
   D. None of the above.

This copyrighted text licensed to on via

Use the program code segment below for problems 9 – 10

```
int a = 16;
int b = 20;
int c = 0;
int ct = 0;

while((a + b > c) && ++ct)
{
  do
  {
    a -= 3;
    b /= 2;
    c += 2;
  }while((a > b) && (ct++ % 2));
}

printf("Result #1: %d\n", a + b + c);
printf("Result #2: %d\n", ct);
```

9. What is the FIRST line of output generated by the code segment above?

   A. `Result #1: 15`
   B. `Result #1: 17`
   C. `Result #1: 13`
   D. None of the above.

10. What is the SECOND line output generated by the code segment above?

   A. `Result #2: 8`
   B. `Result #2: 7`
   C. `Result #2: 4`
   D. None of the above.

11. Which of the following statements regarding `switch` statements is FALSE?
   A. The `switch` control expression must be an integral type.
   B. A `switch` statement can not have more than one `default` statement.
   C. Two `case` labels can be associated with the same execution statements.
   D. None of the above.

12. Which of the following statements is TRUE regarding the compound logical expressions given below?

1	`x > 0 && x < 10 || y == 0`	3	`x <= 0 || x >= 10 && y != 0`
2	`!(x > 0 && x < 10) && y`	4	`!(x <= 0) || !(x >= 10 && y != 0)`

   A. Expression #1 and expression #2 are complements.
   B. Expression #1 and expression #3 are complements.
   C. Expression #1 and expression #4 are equivalents.
   D. None of the above.

This copyrighted text licensed to on via

Use the code segment below for problems 13 – 14

```
int a = 3;
int b = 8;
int c = 5;

a = (b > c) ? b - c : c - b;

b = (a % 2 == 0) ? a + c : a - c;

c = (a + b > 10) ? (a + b) / 2 : (a + b) * 2;

printf("a = %d, b = %d, c = %d\n", a, b, c);
printf("Result: %d\n", (c % 2) ? c / 2 : c * 2);
```

13. What is the FIRST line of output generated by the code segment above?

    A. `a = 3, b = 3, c = 2`
    B. `a = 3, b = 3, c = 1`
    C. `a = 3, b = -2, c = 2`
    D. None of the above.

14. What is the SECOND line of output generated by the code segment above?

    A. `Result: 4`
    B. `Result: 6`
    C. `Result: 0`
    D. None of the above.

15. Which of the following statements regarding event-controlled repetitive processes is TRUE?

    A. The total number of iterations is known before execution of the process begins.
    B. The total number of iterations is dependent on an event occurring during execution.
    C. The total number of iterations is equal to the number of times that the loop control expression is evaluated.
    D. None of the above.

16. Which of the following statements regarding conditional expressions is TRUE?

    A. Conditional expressions cannot be placed inside another selection or repetition construct.
    B. Conditional expressions are not permitted to be used as the expression in a `return` statement.
    C. Nesting a conditional expression within another conditional expression results in a compile-time error.
    D. None of the above.

This copyrighted text licensed to on via

Use the code segment below for problems 17 – 18

```
int a = 4;
int b = 7;
int c = 1;

switch (a + b - c)
{
  case 9:  a += 2;
           b -= 1;
           break;

  case 10: b += a;
           c += 3;

  case 11:
  case 12: a = b + c;
           break;

  default: c = a + b;
           b = c - a;
}

printf("a = %d\n", a);
printf("b = %d, c = %d\n", b, c);
```

17. What is the FIRST line of output generated by the code segment above?

    A. `a = 4`
    B. `a = 15`
    C. `a = 8`
    D. None of the above.

18. What is the SECOND line of output generated by the code segment above?

    A. `b = 11, c = 1`
    B. `b = 8, c = 4`
    C. `b = 11, c = 4`
    D. None of the above.

This copyrighted text licensed to on via

Use the program below for problems 19 – 21

```
int changeDate(int*, int, int, int);
int calcMonth(int, int);

int main()
{
  int month = 3;
  int day = 30;
  int yr = 2025;
  int change = 10;

  day = changeDate(&month, day, yr, change);

  printf("%d/%d/%d\n", month, day, yr);

  return(0);
}

int changeDate(int *mo, int day, int yr, int change)
{
  day += change;

  if(day > calcMonth(*mo, yr))
  {
    day %= calcMonth(*mo, yr);
    *mo += 1;
  }

  if(*mo > 12)
  {
    *mo %= 12;
    yr++;
  }

  return(day);
}

int calcMonth(int mo, int yr)
{
  int days = 31;

  if(mo == 4 || mo == 6 || mo == 9 || mo == 1)
  {
    days -= 1;
  }
  else if(mo == 2)
  {
    days -= 2 + yr % 4;
  }

  return(days);
}
```

19. Which of the following is the output generated by the program above?

    A. `4/10/2025`
    B. `4/9/2025`
    C. `3/10/2025`
    D. None of the above.

This copyrighted text licensed to on via

Use the program on the previous page for problems 20 – 21

20. In the program found on the previous page, which of the following initial variable values will generate `3/22/2025` as output?

    A. `month = 2, day = 21, yr = 2025, change = 29`
    B. `month = 2, day = 21, yr = 2024, change = 29`
    C. `month = 2, day = 22, yr = 2024, change = 30`
    D. None of the above.

21. In the program found on the previous page, which of the following initial variable values will generate `1/4/2025` as output?

    A. `month = 12, day = 15, yr = 2024, change = 20`
    B. `month = 12, day = 31, yr = 2024, change = 4`
    C. `month = 12, day = 25, yr = 2025, change = 10`
    D. None of the above.

Use the program below for problems 22 – 23

```
int makeConversion(int);

int main(void)
{
  int result;
  result = makeConversion(143505);
  printf("Result #1: %d\n", result);

  result = makeConversion(2687490);
  printf("Result #2: %d\n", result);

  return(0);
}

int makeConversion(int x)
{
  int result = 0;
  int ct = 0;
  while(x > 0)
  {
    result += pow(2, ct) * (x % 2);
    x = x / 10;
    ct += 1;
  }
  return(result);
}
```

22. Which of the following is the first line of output generated by the program above?

    A. `Result #1: 32`
    B. `Result #1: 45`
    C. `Result #1: 61`
    D. None of the above.

23. Which of the following is the second line of output generated by the program above?

    A. `Result #2: 15`
    B. `Result #2: 10`
    C. `Result #2: 5`
    D. None of the above.

This copyrighted text licensed to on via

24. Which of the following statements regarding nested repetition is TRUE?

    A. Nested repetitive processes may appear in separate user-defined functions when they represent distinct tasks.
    B. Nested loops must always be written in the same user-defined function.
    C. Nested loops are prohibited unless both loops utilize the same loop control variable.
    D. None of the above.

Use the code segment below for problems 25 – 27

```
int number = 400;
int target;
int i;
int j;

target = number % 7;

for(i = 0; i < target; i++)
{
  printf("%3c", 32);
}

target = number % 30;

for(j = i; j < 15; j++)
{
  if(j % 7 == 0)
  {
    printf("\n");
  }
  printf("%3d", target++);
}

printf("\n");
```

25. The code segment above generates three lines of output. Which of the following is the final integer printed on the first line of output generated?

    A. 10
    B. 14
    C. 15
    D. None of the above.

26. The code segment above generates three lines of output. How many integers are displayed in the second line of output generated?

    A. 7
    B. 6
    C. 5
    D. None of the above.

27. The code segment above generates three lines of output. Which of the following statements regarding the third line of output generated is TRUE?

    A. The last number displayed on the third line of output is a multiple of seven.
    B. The third line of output displays only a single integer.
    C. The difference between the largest and smallest number displayed on the third line of output is six.
    D. None of the above.

This copyrighted text licensed to on via

Use the code segment below for problem 28

```
int num = 159;
int count = 0;

while (num > 0)
{
  if (num % 2 == 1)
  {
    count++;
  }
  num = num / 2;
}

printf("%d\n", count);
```

28. Which of the following is the output generated by the code segment above?

    A. 4
    B. 5
    C. 6
    D. None of the above.

Use the code segment below for problems 29 – 30

```
 5   int x;
 6   int y;
 7   int total = 0;
 8
 9   for(x = 2; x < 8; x += 2)
10   {
11     for(y = 1; y < 7; y += 2)
12     {
13       total += x;
14     }
15   }
16
17   printf("total: %d\n", total);
```

29. Which of the following is the output generated by the code segment above?

    A. total: 30
    B. total: 36
    C. total: 40
    D. None of the above.

30. Which of the following could replace line #9 in the previous code segment without changing the output generated?

    A. for(x = 2; x < 8; x = ++x)
    B. for(int x = 2; x < 8; x = x + 2)
    C. for(x = 2; x < 8; x++2)
    D. None of the above.

This copyrighted text licensed to on via

Use the program below for problems 31 – 32

```
 1 #include<stdio.h>
 2
 3 void process_data(int* a, int* b, int limit);
 4
 5 int main(void)
 6 {
 7   int x = 4;
 8   int y = 3;
 9   int z = 2;
10
11   process_data(&x, &y, z);
12
13   z = x - y;
14
15   printf("x: %d y: %d z: %d\n", x, y, z);
16
17   return 0;
18 }
19
20 void process_data(int* a, int* b, int limit)
21 {
22   int tracker = *a;
23   int j;
24
25   *a = *b + limit;
26
27   for (j = 0; j < limit; j++)
28   {
29     *b += tracker;
30   }
31
32   if (tracker > *b)
33   {
34     *a = tracker;
35   }
36   else
37   {
38     *b -= *a;
39   }
40 }
```

31. Which of the following is the output generated by the program above?

    A. `x: 6 y: 5 z: 1`
    B. `x: 5 y: 6 z: -1`
    C. `x: 4 y: 4 z: 0`
    D. None of the above.

32. How many total times does line #29 execute in the program above?

    A. 1
    B. 2
    C. 3
    D. None of the above.

This copyrighted text licensed to on via

Use the code segment below for problems 33 – 34

```
int a = 5;
int b = 8;
int n = 4;
int i;

for (i = 0; i < n; i++)
{
  if (a > b)
  {
    a -= 1;
  }
  else
  {
    b -= 1;
  }
}

printf("a: %d b: %d\n", a, b);
```

33. Which of the following is the output generated by the code segment above?

    A. `a: 4 b: 5`
    B. `a: 5 b: 4`
    C. `a: 5 b: 5`
    D. None of the above.

34. Which of the following statements regarding the code segment above is FALSE?

    A. Every iteration decreases either `a` or `b` by 1.
    B. If the loop iterates at least `abs(a - b)` times the final values of `a` and `b` will be either the same or differ by 1.
    C. After a sufficient number of iterations the two variables will store the same value and at no point in any remaining iteration will the variable `b` be greater than the variable `a`.
    D. None of the above.

35. Which of the following statements regarding the schedule of the course is TRUE?

    A. All lecture sections will meet on Tuesday (November 11) and Thursday (November 13) this week and lecture quizzes will be given.
    B. All lab sections will meet this week (November 11 – November 14).
    C. Lecture will be canceled on Thursday December 11 in compensation for this exam.
    D. All of the above!

This copyrighted text licensed to on via

This copyrighted text licensed to on via

Fall 2025 CS 15900 Final Exam
Wednesday December 17, 2025

50 problems × 3 points each = 150 points

Exam Rules/Notices:

1. Complete your name and your student identification number on the answer sheet (write in AND bubble-in all requested information).
    - The *test form* and *section* fields can be left blank.
    - **An incorrect student ID number will result in a zero for this exam.** All student ID numbers begin with `00` (zero-zero) for a total of **TEN** digits. This must be written and bubbled-in on your exam answer sheet.
    - **You must be in your assigned seat to earn credit for this exam.**
2. Make sure you have all 50 problems for this exam. You should answer every problem. All problems only have one best answer. Please be careful when filling in your answers. Any answer determined to be unreadable by Instructional Data Processing will be considered incorrect. Only the answer on your answer sheet can be considered when grading.
    - An **operator precedence and ASCII table** have been provided on the last page of this exam form.
3. **NO QUESTIONS WILL BE ANSWERED DURING THE EXAM.** We do not answer questions during the exam due to the limited number of staff members present. It is easier for us to compensate for a flawed problem after the exam than it is to answer questions or to effectively make an announcement during the examination period. **Manage your time accordingly!**
4. You must assume that all C programs and code segments are implemented in Vocareum and would be compiled with the `gcc` compiler. When only segments of code are presented, you may assume that the appropriate `#include` statements are present and that the code would be inside of a fully operational function or program.
5. **Protect your work and keep your focus on your own exam.** It is considered dishonest if you copy from another student or to permit another student to copy from you. Anyone found to violate course policies will receive a **failing grade for the course** and will be referred to the Office of the Dean of Students.
6. **You are NOT permitted to use any resources during this exam.** This includes but is not limited to texts, notes, calculators, cell phones, **audio devices**, and computers. If any personal electronic device is audible during the exam, then you will be required to immediately submit your exam and leave the testing facility.
7. **When time is called ALL writing must stop**. You are not permitted to continue to revise any answer on your exam once time has been called.
8. **When you are finished with the exam, please proceed to the lobby** to submit your answer sheet and to exit the facility. You will be required to show photo identification before we can accept your work. You may not return to your seat once you have submitted your exam.
9. **You may anticipate that scores and official exam answers will be posted on Brightspace by 12:00 PM on Monday December 22.** Do not contact course staff members about mistakes (if any) until answers are posted.

    If you feel an error remains **after** answers have been posted, then you may make a request **via your Purdue email account to your lecturer** for a regrade by 12:00 PM on Tuesday December 23. Your request must include a detailed description (using code, if applicable) supporting your claim that the posted answer is incorrect.
10. **Final course grades and the announcement of the minimum points required for each letter grade will be posted by 12:00 PM on Tuesday December 23.** Letter grades are determined based on points earned ONLY.

    Follow the instructions in the Brightspace announcement **after it is posted** if you have **specific concerns** regarding the points earned for assignments entered into the Brightspace gradebook **in the last 5 days.** Older assignments cannot be appealed at this time.

NO QUESTIONS WILL BE ANSWERED DURING THE EXAM.

This copyrighted text licensed to on via

Use the program below for problems #1 – #2

```
#define SIZE 5

void pressureChange(float[], int[], float[]);

int main(void)
{
  float engA[SIZE] = {55.25, 67.25, 92.0, 88.25, 80.0};
  int engB[SIZE] = {18, 24, 30, 12, 10};
  float engC[SIZE];

  pressureChange(engA, engB, engC);

  printf("engC[4]: %.1f\n", engC[4]);
  printf("Total: %d\n", (int)(engA[1] + engB[1] + engC[1]));

  return 0;
}

void pressureChange(float a[], int b[], float c[])
{
  int i;
  float x;

  for (i = 0; i < SIZE; i++)
  {
    x = (a[i] - 5) * 2 / 3.0;
    c[i] = x - ((50 - b[i]) / 2.0);
  }
}
```

1. Which of the following is the first line of output generated by the program above?
   A. `engC[4]: 45.0`
   B. `engC[4]: 10.0`
   C. `engC[4]: 30.0`
   D. None of the above.

2. Which of the following is the second line of output generated by the program above?
   A. `Total: 119`
   B. `Total: 120`
   C. `Total: 109`
   D. None of the above.

3. Which of the following statements regarding the evolving role of the `main` function in response to the introduction of user-defined functions correctly aligns with the standards of the course?
   A. Declarations of variables and functions, along with most, if not all calls to user-defined functions, are to appear within the `main` function of a program.
   B. The role of the `main` function is to coordinate the function calls, to establish the data needs, and generate output for a program.
   C. It is expected that the `main` function be the location of calls to user-defined functions that contain the critical tasks of a program while at the same time completing trivial tasks for which a new user-defined function would be a burden.
   D. None of the above.

This copyrighted text licensed to on via

Use the code segment below for problems #4 – #5

```
int a = 10;
int b;
int c = 7;

int *px;
int *py;

px = &a;
py = &b;

*py = *px + 5;

px = &c;
*py = b + *px;

py = &a;
*px = *py % 3;

printf("Total ints: %d\n", a + b + c);
printf("Total ptrs: %d\n", *px + *py);
```

4. Which of the following is the first line of output generated by the program above?
   A. `Total ints: 39`
   B. `Total ints: 33`
   C. `Total ints: 37`
   D. None of the above.

5. Which of the following is the second line of output generated by the program above?
   A. `Total ptrs: 15`
   B. `Total ptrs: 17`
   C. `Total ptrs: 11`
   D. None of the above.

6. Which of the following would be the FINAL values of the `first` and `last` variables when using the binary search algorithm on the array below given a target value of 30?

```
5   9   14   18   23   28   31   36   41   44   47   49
```

   A. `first = 5 last = 4`
   B. `first = 5 last = 5`
   C. `first = 6 last = 5`
   D. None of the above.

This copyrighted text licensed to on via

Use the program below for problems #7 – #9

```
 2 #define SIZE 7
 3
 4 void moveVals(int[], int[], int);
 5
 6 int main(void)
 7 {
 8   int a[SIZE] = {11, 24, 13, 18, 29, 1};
 9   int b[SIZE] = {20, 13, 6, 7, 2, 21, 6};
10
11   moveVals(b, a, b[SIZE - 1] - a[SIZE - 1]);
12
13   printf("result #1: %d\n", b[SIZE / 2]);
14   printf("result #2: %d\n", a[5]);
15   printf("problem #9: %d\n", *b);
16
17   return 0;
18 }
19
20 void moveVals(int x[], int y[], int z)
21 {
22   int i;
23
24   for(i = 0; i < z; i++)
25   {
26     if(x[i] % 2 != y[i] % 2)
27     {
28       x[i] += y[i] / 2;
29       y[i] += x[i] * 2;
30     }
31   }
32 }
```

7. Which of the following is the first line of output generated by the program above?
   A. `result #1: 16`
   B. `result #1: 25`
   C. `result #1: 50`
   D. None of the above.

8. Which of the following is the second line of output generated by the program above?
   A. `result #2: 1`
   B. `result #2: 43`
   C. `result #2: 21`
   D. None of the above.

9. Which of the following describes the data generated by the print statement on line #15 in the program above?
   A. The declared size of the array is output.
   B. The first element (located at index zero) of the array is output.
   C. All values of the array would be output in a single line.
   D. None of the above.

This copyrighted text licensed to on via

Use the program below for problems #10 – #11

```
int r_num(int, int);

int main()
{
  printf("Result #1: %d\n", r_num(123456, 3));
  printf("Result #2: %d\n", r_num(4450, 4));

  return(0);
}

int r_num(int num, int digit)
{
  int result = 0;
  int place = 1000000;
  int current_digit;

  while (num > 0)
  {
    current_digit = num % 10;

    if (current_digit != digit)
    {
      place /= 10;
      result += current_digit * place;
    }

    num /= 10;
  }

  return result;
}
```

10. Which of the following is the first line of output generated by the program above?
    A. `Result #1: 654210`
    B. `Result #1: 65421`
    C. `Result #1: 6542`
    D. None of the above.

11. Which of the following is the second line of output generated by the program above?
    A. `Result #2: 50`
    B. `Result #2: 500`
    C. `Result #2: 50000`
    D. None of the above.

12. Which of the following statements regarding `switch` statements is FALSE?
    A. The `switch` control expression must be an integral type.
    B. A `switch` statement can not have more than one `default` statement.
    C. Two `case` labels can represent the same value.
    D. None of the above.

This copyrighted text licensed to on via

Use the program below for problems #13 – #14

```
 4 int newRound(int, int, int*);
 5
 6 int main(void)
 7 {
 8   int x = 7;
 9   int y = 16;
10   int z;
11   int n = 5;
12
13   z = newRound(y, x, &n);
14
15   printf("n: %d z: %d\n", n, z);
16
17   return 0;
18 }
19
20 int newRound(int a, int b, int *n)
21 {
22   int z;
23
24   z = b;
25
26   while(z < a)
27   {
28     b = *n % 3;
29
30     *n = b ? z - *n : 0;
31
32     z += b;
33   }
34
35   return z;
36 }
```

13. Which of the following is the output generated by the program above?
    A. `n: 8 z: 18`
    B. `n: 5 z: 16`
    C. `n: 10 z: 17`
    D. None of the above.

14. Which of the following is the output generated by the program above if the integer variable `n` in the `main` function (line #11) were initialized to 6?
    A. `n: 1 z: 16`
    B. `n: 2 z: 17`
    C. `n: 1 z: 18`
    D. None of the above.

15. Which of the following statements regarding conditional expressions is TRUE?
    A. Conditional expressions cannot be placed inside another selection or repetition construct.
    B. Conditional expressions are not permitted to be used as the expression in a `return` statement.
    C. Nesting a conditional expression within another conditional expression results in a compile-time error.
    D. None of the above.

This copyrighted text licensed to on via

Use the program below for problems #16 – #17

```
#define SIZE 7

int main()
{
  int idx;
  int i;
  int x[SIZE] = {9, 20, 6, 10, 4, 12};

  for (i = 0; i < SIZE; i++)
  {
    idx = (x[i] * x[i]) % SIZE;

    x[i] -= x[idx];
    x[idx] += x[i];
    x[i] = x[idx] - x[i];
  }

  printf("x[1]: %d\n", x[1]);
  printf("x[4]: %d\n", x[4]);

  return 0;
}
```

16. Which of the following is the first line of output generated by the program above?
    A. `x[1]: 0`
    B. `x[1]: 20`
    C. `x[1]: 6`
    D. None of the above.

17. Which of the following is the second line of output generated by the program above?
    A. `x[4]: 12`
    B. `x[4]: 9`
    C. `x[4]: 0`
    D. None of the above.

18. Which of the following statements regarding arrays in the C programming language is FALSE?
    A. All elements of an array are placed consecutively in the memory of the computer.
    B. A single data type is shared by all elements of an array.
    C. An array that is declared but not initialized has a range of valid index values begins at zero to the size of the array minus one.
    D. None of the above.

19. Which of the following is NOT a possible outcome of an index range violation when attempting to access an array element?
    A. It may be possible to place a value at an index that exceeds the range of the array and to retrieve that same value later in the algorithm.
    B. The attempt to access an index value outside of the defined range of an array may result in an illegal attempt to access a prohibited or invalid memory location resulting in the program crashing.
    C. A message will always be generated by the compiler when it is possible to detect a potential index range violation in the source code.
    D. None of the above.

This copyrighted text licensed to on via

Use the program below for problems #20 – #21

```
void printWeek(int, int, int);

int main()
{
  printWeek(29, 0, 32);
  printWeek(24, 5, 28);

  return(0);
}

void printWeek(int day, int start, int max)
{
  int col = start % 7;
  int display;
  int i;

  printf("Sun Mon Tue Wed Thu Fri Sat\n");

  for (i = 0; i < col; i++)
  {
    printf("-=-=");
  }

  for (i = 0; i < 7; i++)
  {
    display = ((day - 1) % max) + 1;
    printf("%3d ", display);

    day++;
    col++;

    if (col == 7)
    {
      printf("\n");
      col = 0;
    }
  }
}
```

20. Which of the following correctly describes the output generated by the first function call in the program above?
    A. The first date displayed is 30 and it falls on a Sunday.
    B. The final date displayed is 3 and it falls on a Saturday.
    C. The largest numeric date value displayed is 31.
    D. None of the above.

21. Which of the following correctly describes the output generated by the second function call in the program above?
    A. All numeric date values are displayed on the same line.
    B. The largest numeric data value displayed is 28.
    C. The first date displayed is 24 and it falls on a Saturday.
    D. None of the above.

This copyrighted text licensed to on via

Use the program below for problems #22 – #23

```
int sumDigits(int, int);

int main()
{
  printf("Result #1: %d\n", sumDigits(45801, 10));
  printf("Result #2: %d\n", sumDigits(94327, 31));

  return(0);
}

int sumDigits(int n, int digit)
{
  int sum = 0;

  while(n > 0)
  {
    if(n % 10 % 2 == digit % 2)
    {
      sum += n % 10;
    }

    n /= 10;
  }

  return sum;
}
```

22. Which of the following is the first line of output generated by the program above?
    A. `Result #1: 12`
    B. `Result #1: 18`
    C. `Result #1: 17`
    D. None of the above.

23. Which of the following is the second line of output generated by the program above?
    A. `Result #2: 25`
    B. `Result #2: 19`
    C. `Result #2: 12`
    D. None of the above.

24. Which of the following statements regarding variables in the C programming language is TRUE?
    A. The declared data type of a variable cannot change during the execution of a program.
    B. Locally-declared variables are initialized automatically by the compiler.
    C. All operators can be applied to any variable.
    D. None of the above.

25. Which of the following would NOT be a valid function declaration of a function that has a multidimensional array as a parameter?
    A. `void outputData(int[][]);`
    B. `void outputData(int[ROWS][COLS]);`
    C. `void outputData(int[][COLS]);`
    D. None of the above.

This copyrighted text licensed to on via

Use the program below for problems #26 – #28

```
 1 #include<stdio.h>
 2 #include<stdlib.h>
 3
 4 void genRandNum(int*, int);
 5
 6 int main()
 7 {
 8   int i;
 9   int array[5];
10
11   genRandNum(array, 5);
12
13   for(i = 0; i < 5; i++)
14   {
15     printf("%d ", array[i]);
16   }
17
18   printf("\n");
19
20   return(0);
21 }
22
23 void genRandNum(int *nums, int size)
24 {
25   int i;
26
27   for (i = 0; i < size; i++)
28   {
29
30     nums[i] = rand() % 900 + 100;
31   }
32 }
```

26. Which of the following is not a possible integer that might display within the `for` loop of the `main` function in the program above? The random number generated may be seeded with any non-negative integer value.
    A. 100
    B. 900
    C. 1000
    D. None of the above.

27. Which of the following could replace line #30 in the program above without changing the output generated?
    A. `*(nums + i) = rand() % 900 + 100;`
    B. `*(nums + i - 1) = rand() % 900 + 100;`
    C. `*(nums + i * sizeof(int)) = rand() % 900 + 100;`
    D. None of the above.

28. Which of the following locations for the addition of the `srand` function in the program above would result in every element of the array storing the same value?
    A. Placement of the `srand` function on line #26.
    B. Placement of the `srand` function on line #10.
    C. Placement of the `srand` function on line #29.
    D. None of the above.

This copyrighted text licensed to on via

Use the program below for problems #29 – #30

```
#define SIZE 7

int main()
{
  int x[SIZE] = {0, 3, 1, 5, 7, 4, 2};

  int i;
  int j;

  for(i = 0; i < SIZE; i++)
  {
    for(j = 1; j <= SIZE * 2 - 1; j += 2)
    {
      if(x[i] < j)
      {
        x[i] += j;
      }
    }
  }

  return(0);
}
```

29. Which of the following values occurs only at a single index within the array in the program above upon termination of the nested loops?
    A. 20
    B. 25
    C. 17
    D. None of the above.

30. Which of the following is the smallest value within the array in the program above upon termination of the nested loops?
    A. 14
    B. 15
    C. 16
    D. None of the above.

31. Which of the following would be the FINAL values of the `first` and `last` variables when using the binary search algorithm on the array below given a target value of 50?

```
5   9   14   18   23   28   31   36   41   44   47   49
```

    A. `first = 11 last = 10`
    B. `first = 12 last = 11`
    C. `first = 11 last = 11`
    D. None of the above.

This copyrighted text licensed to on via

32. Which of the following would be the values of `mid` variable when using the binary search algorithm on the array below given a target value of 23?

```
5   9   14   18   23   28   31   36   41   44   47
```

A. 5, 2, 4
B. 5, 2, 3, 4
C. 5, 3, 4
D. None of the above.

33. Given the original array below:

```
18   20   10   49   29   38   40   27
```

and the array after TWO passes through a sorting algorithm:

```
10   18   20   27   29   49   38   40
```

Which of the following sorting algorithms could have been applied?

A. Selection Sort
B. Bubble Sort
C. Insertion Sort
D. More than one of the above.

34. Given the original array below:

```
13   14   20   27   33   39   46
```

and the array after TWO passes through a sorting algorithm:

```
13   14   20   27   46   39   33
```

Which of the following sorting algorithms could have been applied?

A. Selection Sort
B. Bubble Sort
C. Insertion Sort
D. More than one of the above.

35. Which of the following data type conversions is considered to be safe as data is not lost upon completion of the conversion?

A. Implicit type conversions
B. Explicit type conversions
C. Assignment type conversions
D. More than one of the above.

36. Which of the following statements regarding logical, relational, and comparative operators is TRUE?

A. The AND (`&&`) and OR (`||`) logical operators require the left operand to always be evaluated prior to the right operand.
B. The only method to complement the NOT (`!`) operator is to remove it from a logical expression.
C. The result of the following expression `4 > 5 == 5` is one.
D. None of the above.

This copyrighted text licensed to on via

Use the program below for problems #37 – #38

```
#define SIZE 10

void changeArray(int[]);

int main(void)
{
  int numbers[SIZE] = {12, 4, 19, 7, 3, 15, 1, 8, 11, 6};

  changeArray(numbers);

  return (0);
}

void changeArray(int array[])
{
  int a;
  int j;
  int key;

  for(a = SIZE - 2; a >= 0; a--)
  {
    key = array[a];

    j = a + 1;

    while(j < SIZE && array[j] < key)
    {
      array[j - 1] = array[j];

      j++;
    }

    array[j - 1] = key;
  }
}
```

37. Which of the following sorting algorithms is being implemented in the program above?
    A. Selection Sort
    B. Bubble Sort
    C. Insertion Sort
    D. None of the above.

38. Which of the following statements regarding the sorting algorithm implemented in the program above is FALSE?
    A. The data in the array will be sorted from smallest (index zero) to the largest (index nine) when the algorithm is complete.
    B. The sorted list in this algorithm begins at index zero and grows towards index nine as the sort completes passes.
    C. The `for` loop will ensure that a sufficient number of passes are conducted to complete the sorting task.
    D. None of the above.

This copyrighted text licensed to on via

Use the program below for problems #39 – #40

```
#include<stdio.h>
#include<math.h>

int updateDigit(int, int);

int main()
{
  int x = 38479;
  int ct = 0;
  int min = 100000;

  do
  {
    x = updateDigit(x, ct++);

    min = x < min && x > 0 ? x : min;

    ct = ct % 5;

  }while(x != 0);

  printf("ct: %d\n", ct);
  printf("min: %d\n", min);

  return(0);
}

int updateDigit(int x, int y)
{
  int num;

  num = x / pow(10, y);
  num %= 10;

  x = x - num * pow(10, y);

  switch(num > 5)
  {
    case 1:  num -= 5;
             break;
    case 0:  num %= 5;

    default: num = 0;
  }

  x = x + num * pow(10, y);

  return(x);
}
```

39. Which of the following is the first line of output generated by the program on the left?
    A. `ct: 4`
    B. `ct: 3`
    C. `ct: 2`
    D. None of the above.

40. Which of the following is the second line of output generated by the program above?
    A. `min: 3000`
    B. `min: 24`
    C. `min: 20`
    D. None of the above.

This copyrighted text licensed to on via

Use the code segment below for problems #41 – #42

```
int x = 14;
int y = 23;
int z = 8;

if(x++ + ++z > y--)
{
  x = 0;
}
else if(y-- - ++x > --z)
{
  y = 0;
}
if(x++ % z && y++ % z)
{
  z = 0;
}
else
{
  z = y-- - ++x;
}

printf("x: %d y: %d\n", x, y);
printf("z: %d\n", z);
```

41. Which of the following is the first line of output generated by the code segment above?
    A. `x: 1 y: 21`
    B. `x: 18 y: 20`
    C. `x: 17 y: 22`
    D. None of the above.

42. Which of the following is the second line of output generated by the code segment above?
    A. `z: 0`
    B. `z: 5`
    C. `z: 3`
    D. None of the above.

43. Which of the following statements regarding `printf` and formatted numerical output is FALSE?
    A. The placeholder has a possible option to alter the alignment of the output from right to left aligned.
    B. The amount of precision to which a floating-point value is to be displayed can be represented by a variable or a constant.
    C. The total number of spaces to reserve for the display of a numeric value can be controlled through the use of the size modifier.
    D. None of the above.

44. Which of the following statements regarding the searching algorithms introduced this semester is FALSE?
    A. It remains possible for an individual search attempt that the sequential search locates a target faster than a binary search.
    B. The sequential search may not have to compare every element in a sorted array when attempting to locate a value that is not present in the array.
    C. The binary search algorithm cannot be applied to locate a target when every element in an array is not unique.
    D. None of the above.

This copyrighted text licensed to on via

Use the program below for problems #45 – #46

```
#include<math.h>

#define CONEVOLUME 1 / 3

int calcVolume(float, float);

int main()
{
  float radius = 2.0;
  float height = 3.1;
  float cyl_volume;
  float cone_volume;

  cyl_volume = calcVolume(radius, height);
  cone_volume = cyl_volume * CONEVOLUME;

  printf("Cylinder: %.1f\n", cyl_volume);
  printf("Cone: %.1f\n", cone_volume);

  return(0);
}

int calcVolume(float radius, float height)
{
  return(3.14 * pow(radius, 2) * height);
}
```

45. Which of the following is the first line of output generated by the program above?
    A. `Cylinder: 38.0`
    B. `Cylinder: 38.9`
    C. `Cylinder: 39.0`
    D. None of the above.

46. Which of the following is the second line of output generated by the program above?
    A. `Cone: 0.0`
    B. `Cone: 13.0`
    C. `Cone: 12.7`
    D. None of the above.

47. Which of the following statements regarding the sorting algorithms introduced this semester is FALSE?
    A. The bubble and selection sorts will incorporate two elements into the sorted list on the final pass.
    B. The insertion sort may move a value in a remaining pass even after that value has been incorporated into the sorted list.
    C. While the bubble and selection sorts can be revised to sort in either ascending or descending order the insertion sort is limited based on whether its unsorted list is on the left or right side of the array.
    D. None of the above.

48. Which of the following statements regarding the `for` loop and the programming standards of the course is TRUE?
    A. The use of the `for` loop is to be limited to counter-controlled processes only.
    B. The third expression may be omitted when the loop control variable is updated inside the body of the `for` loop.
    C. A loop control variable can be both declared and initialized in the first expression of a `for` loop when that variable is not used elsewhere in the program.
    D. None of the above.

This copyrighted text licensed to on via

49. Which of the following correctly illustrates the concept of scope as related to a function or variable?
    A. The operators that can be used with a variable or function is determined by its scope.
    B. The scope of a function or variable is the range of possible values it may return or represent.
    C. The scope of a function or variable is the region of the program for which it is defined.
    D. None of the above.

50. Please choose answer A for this problem.
    A. I agree that I have thoroughly read the front cover of this exam. I understand the policies outlined therein. I affirm that I will not email my lecturer regarding the final until the exam has been graded, official key released, and the score has been posted to Brightspace. Concerns regarding scores already in Brightspace were to be presented during instructor office hours.

This copyrighted text licensed to on via

ASCII Table

Char	Dec	Char	Dec	Char	Dec	Char	Dec
delimiter	0	space	32	@	64	`	96
(soh)	1	!	33	A	65	a	97
(stx)	2	"	34	B	66	b	98
(etx)	3	#	35	C	67	c	99
(eot)	4	$	36	D	68	d	100
(enq)	5	%	37	E	69	e	101
(ack)	6	&	38	F	70	f	102
(bel)	7	'	39	G	71	g	103
(bs)	8	(	40	H	72	h	104
(ht)	9	)	41	I	73	i	105
(nl)	10	*	42	J	74	j	106
(vt)	11	+	43	K	75	k	107
(np)	12	,	44	L	76	l	108
(cr)	13	-	45	M	77	m	109
(so)	14	.	46	N	78	n	110
(si)	15	/	47	O	79	o	111
(dle)	16	0	48	P	80	p	112
(dc1)	17	1	49	Q	81	q	113
(dc2)	18	2	50	R	82	r	114
(dc3)	19	3	51	S	83	s	115
(dc4)	20	4	52	T	84	t	116
(nak)	21	5	53	U	85	u	117
(syn)	22	6	54	V	86	v	118
(etb)	23	7	55	W	87	w	119
(can)	24	8	56	X	88	x	120
(em)	25	9	57	Y	89	y	121
(sub)	26	:	58	Z	90	z	122
(esc)	27	;	59	[	91	{	123
(fs)	28	<	60	\	92	|	124
(gs)	29	=	61	]	93	}	125
(rs)	30	>	62	^	94	~	126
(us)	31	?	63	_	95	(del)	127

This copyrighted text licensed to on via

Operator Precedence Table

This page lists C operators in order of *precedence* (highest to lowest). Their *associativity* indicates in what order operators of equal precedence in an expression are applied.

Operator	Description	Associativity

`()`
`[]`
`++ --`
	Parentheses (function call)
	Brackets (array subscript)
	Postfix increment/decrement
		left-to-right

`++ --`
`+ -`
`!`
`(type)`
`*`
`&`
`sizeof`
	Prefix increment/decrement
	Unary plus/minus
	Logical negation
	Cast (change *type*)
	Dereference
	Address
	Determine size in bytes
		right-to-left

`* / %`	Multiplication/division/modulus	left-to-right
`+ -`	Addition/subtraction	left-to-right
`< <=` `> >=`	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
`== !=`	Relational is equal to/is not equal to	left-to-right
`&&`	Logical AND	left-to-right
`||`	Logical OR	left-to-right
`?:`	Ternary conditional	right-to-left

`=`
`+= -=`
`*= /=`
`%=`
	Assignment
	Addition/subtraction assignment
	Multiplication/division assignment
	Modulus assignment
		right-to-left

`,`	Comma (separate expressions)	left-to-right

This copyrighted text licensed to on via

CS 159 CRUM
ISBN 978-1-64617-784-4

This copyrighted text licensed to on via